# 6809マイコン・システム設計作法

リアルタイム・モニタ/組み込みコンピュータのための

鶴見惠一 著

●良い製品には良い設計思想が生かされており，良い設計思想には，必ずしっかりとしたバックグラウンドがあります．

●ますますIC/LSI化が進み高度化するエレクトロニクス技術も，バックグラウンドとなる技術さえあれば怖くありません．

●CORE BOOKSは，あなたのエレクトロニクス技術のバックグラウンドづくりを応援するCQ出版社の新しい書籍シリーズです．

〈カバー〉
●デザイン／宰　良二
●フォト　／佐瀬　真

# 6809マイコン・システム設計作法

## リアルタイム・モニタ/組み込みコンピュータのための

鶴見惠一 著

CQ出版社

# まえがき

我が国ではファンの少なかった68系のマイクロプロセッサも，究極の8ビット・プロセッサと呼ばれながら6809が登場して以来，その優れたアーキテクチャと良く整理されたインストラクションとが注目され，すっかり定着して強力な地位を築きつつあるように思います．

さらに現在では，16ビット・プロセッサが急速に普及しつつあり，68000は16ビットの代表と呼ぶにふさわしいパフォーマンスを秘めています．

しかし残念ながら，どちらもZ80や8086に遅れて発表されたためか，パソコンに使用される機会が少なかったため，普及度においてはだいぶ遅れをとっているようです．

そこで本書では，68系の良いアーキテクチャをできるだけ身近な物としていただけるように，システムを設計するうえで最低必要な実例を示しながら基礎的な説明に努めました．

ここでは，68系の8ビットを代表する6809についてその応用を紹介しますが，6809については68000を意識した説明とし，将来16ビットへ移行する際にも参考になるよう努めたつもりです．

本書より，一人でも多くの方が優れた特長をもつ68系プロセッサに関心を抱かれ，この応用技術がさらに発展するよう願ってやみません．

1987年4月　著者

# 目　　次

〈カバー〉デザイン/辛 良二
フォト/佐瀬 真

# 第1章

# 6809のアーキテクチャ

6809は6800の上位プロセッサとして誕生したのですが，8080Aに対するZ80の上位とはその意味合いが少し異なります．結論を先にあげれば8080AとZ80とでは機械語において上位互換性がありますが，6800と6809ではこの互換性がないのです．

6809の6800に対する上位互換性とは，ソース・レベルにおいてのみ原則として上位互換性が保たれています．原則といったのはアセンブラのソースであっても厳密にいえば完全な上位互換ではないのです．プログラムによっては一部の修正を必要とする場合があります．

この一部とは特別の場合と考えてよいので，6800のプログラムはアセンブルし直おせば6809で再利用可能であると，ほとんどの場合にいえますが，いずれにしてもこの互換性についてはZ80に一歩を譲らねばなりません．

ではなぜ，6800で蓄積したソフトウェアの再利用に不自由が生じる，というような覚悟をしたうえでのアーキテクチャとなったのでしょう．

筆者が直接に開発スタッフと議論したわけではないので，推測の域を脱しませんが次のようなことがいえると思います．

6800と6809とでは，プロセッサの開発時期を考えた場合，その応用される環境が大きく変化し，進歩したこと，従って，たんなる機能の拡張というだけでは不満であったと考えます．さらに6800の機械語命令をまったくそのままに温存して，そのうえに6809のインストラクションを構築したとしたら，私たちが現在見るような効率の良さと大変良く整理されたアーキテクチャの実現には致らなかったように思うのです．

このように，重大な問題でもある完全上位互換性を放棄することで，一方では開発時点で理想とする概念で妥協することなく新しいプロセッサを誕生させたと想像します．このことが6809を究極の8ビット・プロセッサと呼ばせる結果となったのでしょう．

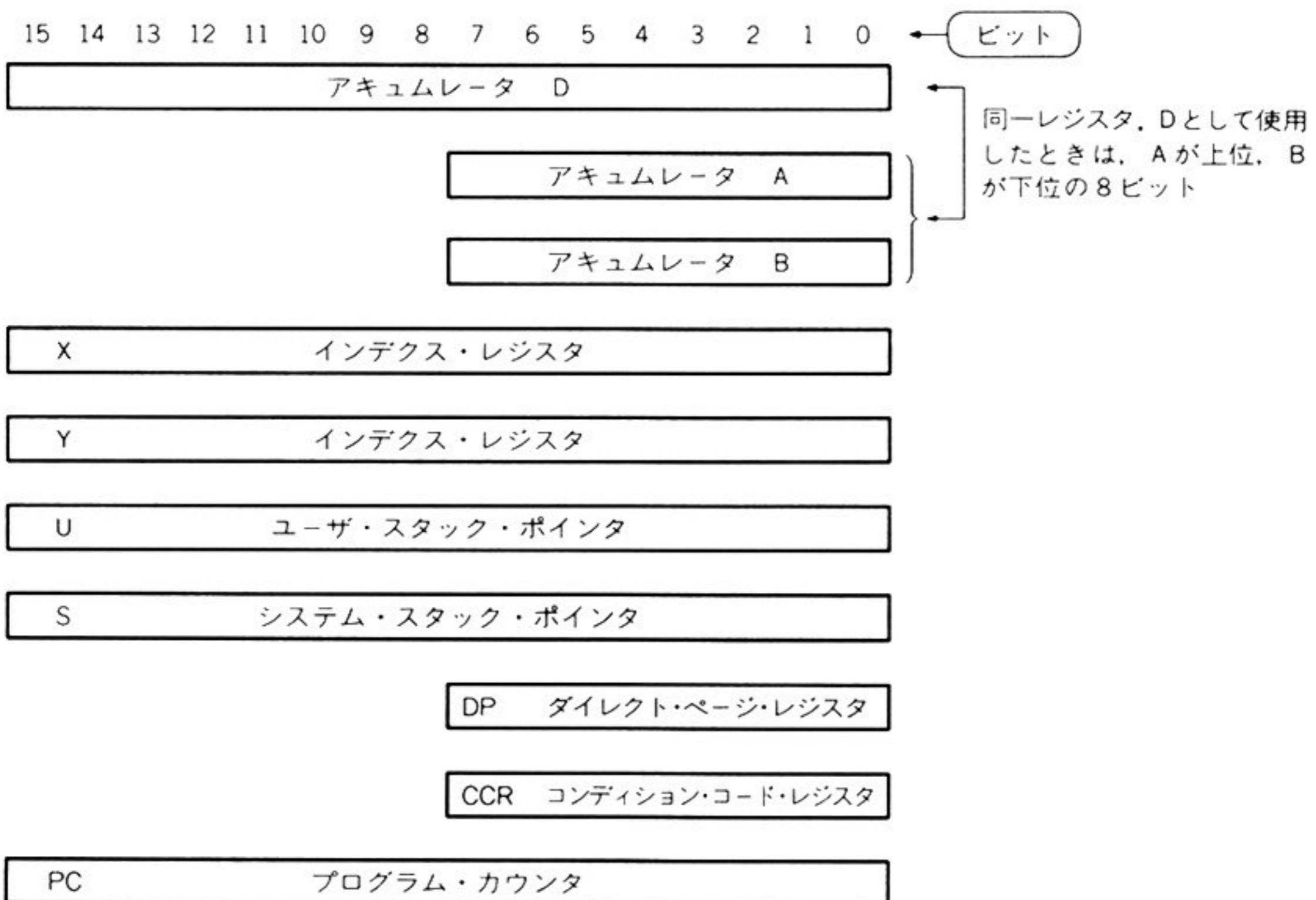
6809の特徴をあげればきりがないのですが，6809に馴染みの薄い読者のために，筆者が感じるままの一部を紹介しておきます．

8ビットのアキュムレータを二本もったダブル・アキュムレータ構成という点では6800と同様であり，68系の特徴でもありますが，6809ではこの二つのアキュムレータを連結して16ビットのレジスタとして使用することができます．これだけではありませんが16ビット・データの処理では格段の強みを発揮します．

二本のスタック・ポインタをもっているために，スタックを利用した高度のプログラミング・テクニックを駆使することが可能です．アドレシング・モードが大幅に拡張され，どんな場合でもアドレシングの不自由は感じません．

全体として，リエントラント可能なプログラム，高級言語のサポート，構造化プログラミングといった問題に見事に対応した，真に現代のプロセッサであると思います．

マイクロプロセッサの呼び名として“CPU”が一般的に使われていますが，モトローラでは“MPU”（マイクロ・プロセシング・ユニット）と呼んでいます．ここでもMPUと呼ぶことにしました．

## 1.1 レジスタ構成とレジスタの機能

6809の内部レジスタを図1.1に示します．この構成で，レジスタが少ないと見るか多いと見るかは，これまでにどのようなプロセッサに馴染んできたのか，あるいはどのようなプログラミングの癖があるのかによって異なると思いますが，使いこむにつれてそれぞれのレジスタが目的にそって，よく機能する過不足のない構成であるように思えてきます．

**● アキュムレータ(A，B，D)**

アキュムレータはAとBの二本があります．これは演算処理の実行に使用される汎用レジスタであり，AとBは一部の命令を除いて同様の機能をもっています．一部の命令とは次のものです．

10進数の加算後における調整であるDAA(デシマル・アジャスト)はAレジスタだけが可能であり，Bレジスタだけが可能な命令としてABX(XとBの加算)があります．

Dレジスタは独立して存在するのではなく，AとBを連結して16ビット・レジスタとして使用可能であり，この場合にDレジスタと呼ばれます．このときAは上位バイト，Bは下位バイトに配置されます．

図1.1[1] 6809の内部レジスタ

● ダイレクト・ページ・レジスタ(DP)

ダイレクト・アドレシングが指定された場合，アドレスの上位8ビットを指定するレジスタです．下位の8ビットは命令のオペランドで指定します．

メモリの限られた部分を頻繁にアクセスする場合には，このレジスタの使用により実行スピードの向上とプログラム・サイズの短縮化を図ることができます．

● インデクス・レジスタ(X，Y)

16ビット長のインデクス・レジスタを二本備えています．インデクスト・アドレシング・モードで使用されますが，自動的にインクリメントまたはデクリメントすることが可能であり，スタック・ポインタ的な使用もできます．

これらのレジスタはアドレスの指示に使用されるため，次のU,Sを含めてポインタ・レジスタとも呼ばれます．

● スタック・ポインタ(U，S)

スタック・ポインタもUとSの二本が用意されています．Sはシステム・スタック・ポインタであり，割り込み時やサブルーチン呼出しでの自動的な内部レジスタの退避と復帰

図1.2[2] コンディション・コード・レジスタのフラグの配列

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |

C：キャリ　：最上位ビットからの桁上げを示す．減算によるボローもこのビットで示す

V：オーバフロー　：符号付き2の補数表現によるオーバフローを示す

Z：ゼロ　：結果がゼロを示す

N：ネガティブ　：2の補数表現による負．すなわち最上位ビットが1のときセット

I：$\overline{\mathrm{IRQ}}$マスク　：このビットが1であれば，$\overline{\mathrm{IRQ}}$はマスク

H：ハーフ・キャリ　：8ビット加算の結果，ビット3からのキャリを示す．BCD演算で必要

F：$\overline{\mathrm{FIRQ}}$マスク　：このビットが1であれば，$\overline{\mathrm{FIRQ}}$はマスク

E：エンタイア・フラグ：割り込みによりすべてのレジスタが退避されていることを示す．NMI，IRQでは1にセット，FIRQでは0になる

すべてのビットは正論理であり，1のとき上に示した状態，0ではそうでないことを示す

に使用されます．Uはユーザ・スタック・ポインタと呼ばれ，完全にユーザに解放されたスタック・ポインタです．UとSは，X，Yと同様にインデクス・レジスタとしても使用できます．

● **プログラム・カウンタ(PC)**

命令のアドレスを示すレジスタですが，これも6809ならではの特徴をもっています．まず，インデクス・レジスタとして機能させることができ，プログラム・カウンタ相対アドレシングとして，プログラム・カウンタとオフセット値によるアドレシングが可能です．すなわち完全なポジション・インディペンデントのプログラムが大変容易に記述できます．

PCは，ほかの16ビット・レジスタと交換および転送が可能であり，演算による流れの変更，すなわちコンピューテッドGOTOが容易に実現できます．

● **コンディション・コード・レジスタ(CC)**

MPUの実行の結果または状態を示すフラグ・レジスタです．各ビットはそれぞれフラグとして割り当てられ，その配列は図1.2を参照してください．各ビットの意味を説明します．

(C)　キャリ

演算の結果が，最上位ビットからの桁上げを生じた場合にセットされます．減算命令によるボローを示すのにもこのビットが使われます．

(V)　オーバフロー

符号付き2の補数表現によるデータがオーバフローした場合，つまり8ビットでは+127，−128を超えた場合，16ビットでは+32767，−32768を超えた場合にセットされます.

(Z)　ゼロ

データ処理の結果がゼロのときにセット("1")されます.

(N)　ネガティブ

データ処理の結果，最上位ビットが1のとき，すなわち2の補数表現では負のときセットされます.

(I)　$\overline{\text{IRQ}}$マスク

このビットがセットされていれば，$\overline{\text{IRQ}}$はマスクされ受け付けられません．割り込みやリセットによる起動がかかると，このビットはセットされますが，ソフトウェア割り込みのSWI2とSWI3は例外であり，このビットに影響をあたえません.

(H)　ハーフ・キャリ

8ビット加算の結果，ビット3からのキャリがセットされます．BCD加算の補正処理を行うDAA命令で使用されます．このビットは加算命令であるADC，ADDの2命令でのみ意味をもちます.

(I)　$\overline{\text{FIRQ}}$マスク

このビットがセットされていれば，$\overline{\text{FIRQ}}$はマスクされ受け付けられません．$\overline{\text{NMI}}$，$\overline{\text{FIRQ}}$，$\overline{\text{RES}}$信号による起動およびSWIによる起動では自動的にセットされますが，SWI2，SWI3による割り込みはこのビットに影響を与えません.

(E)　エンタイア・フラグ

割り込みが起動したとき，すべてのレジスタが退避された場合にセットされます．"0"はPCとCCのみが退避されていることを示します．NMI，IRQによる割り込みでは"1"にセットされますが，FIRQでは"0"になります.

### コンピューテッド GOTO

プログラムの流れを分岐させる場合にはブランチ命令が使われますが，いくつか用意された分岐先の中から，どこへ分岐するかを実行時の演算結果によって決定することをコンピューテッドGOTOと呼びます．BASICではON…GOTOがこれに相当します.

## 1.2 アドレシング・モード

6809に初めて接した方は，アドレシング・モードの多さに驚かれることと思います．これが6809の最も優れた点でもあるのですが，そのため6809が難解に見えることも否定できません．

しかし，これらの全部を理解しなければプログラムできないというわけではないのです．まずは理解したモードだけを使用してプログラムに挑んでください．そしてしばしばこのページにもどって，少しずつレパートリを増やしていけばよいでしょう．

6809が現代的であるとは，これらの強力なアドレシング・モードによって，リエントラントなプログラムやリロケータブルなプログラム，さらにはプログラムのブロック化などが容易に行えるためでもあります．

アドレシング・モードは，全部で10種類に分類することができます．それぞれについて順に説明します．

**● インヘレント・アドレシング**

このモードによる命令では，オペコードにアドレス情報がすべて含まれています．従って，オペランドを取りません．

```
(例)    ABX
        ASRA
        CLRA
```

ABXはAレジスタとXレジスタの加算であり，ASRAはAレジスタのシフト命令です．すなわち，どのデータをどうするのかということがオペコードにすべて含まれてしまっているのが，インヘレント・アドレシングです．

**● イミディエイト・アドレシング**

オペランド部で示される値そのものがデータとなります．アセンブラでは#がそのことを示します．

```
(例)    CMPA   #$41
        LDX    #DATA1
        LDU    #$2000
```

図1.3
エクステンデド・インダイレクト・アドレシングの例

CMPAの例では，Aレジスタと$41が比較されます．つぎのLDXの例では，ラベルDATA1で示されるアドレスの内容ではなく，アドレス値そのものがXレジスタにロードされます．

● エクステンデド・アドレシング

16ビットのフル・アドレスで実効アドレスを指定します．絶対アドレス指定であり，目的とするアドレスがそれ自身のプログラム内である場合には再配置可能ではないので注意してください．

ダイレクト・アドレシングと比較して，その違いを理解してください．

```
(例)    CMPA   $0041
        LDX    DATA1
        LDU    $2000
```

CMPAの例では，アドレス値$41のメモリ内容と比較が行われます．LDXの例ではDATA1で示すメモリの内容がXレジスタにロードされます．上記のイミディエイト・アドレシングと明確に区別してください．

● エクステンデド・インダイレクト・アドレシング

メモリ間接モードと呼ばれるアドレシングです．オペランドの値はメモリ・アドレスを示しますが，その値はさらに目的とするデータのアドレスを示します．この例は図1.3を参

照してください.

このアドレシングでは，メモリをインデクス・レジスタとして使用することができます．6809の特徴的なアドレシング・モードであり，68000にはこのモードはありません.

```
(例)  LDX   [LABEL1]
      LDA   [$1000]
```

●ダイレクト・アドレシング

命令のオペランド部は1バイトであり，オペランドはアドレスの下位8ビットを指定します．上位8ビットはダイレクト・ページ・レジスタが受けもちます．このためエクステンデド・アドレシングに比べ，命令語長が1バイト短く，実行速度も速くなります.

```
(例)  LDA   #$E1
      TFR   A, DP
      LDD   <$E110
```

<はアセンブラに対し，ダイレクト・アドレシングを指定します．この例では，オペランド値が16進4桁で書かれていますが，アセンブラがダイレクト・アドレシングの指定を受け入れた場合には，上位バイトは無視されます.

**● レジスタ・アドレシング**

オペランドはMPUの内部レジスタを指定します．このモードを使用する命令は次のとおりです.

| TFR, EXG, PSHS, PSHU, PULS, PULU |
|---|

```
(例)  TFR   A, DP
      EXG   X, PC
      PSHS  D, X, U
```

TFRの例では，AをDPに転送します．EXGの例はXとPCを交換します．PSHSではD, X, UをスタックSに退避します.

**● インデクスト・アドレシング**

このモードも6809を特徴づけるアドレシングです．6800とは比較にならないほど強力なものであり68000に匹敵します．ですから，このアドレシング・モードの理解は，68000の理解も容易にします.

このモードでは，ポインタ・レジスタ(X，Y，UまたはS)と指定されたオフセット値に基づき，実効アドレスが命令の実行時に計算され，メモリのアクセスを行います．さらにオフセット値の指定方法やオート・インクリメント/デクリメントが加わり，5種類のモードに分類されます．

以下それぞれについて順に説明します．

(1) ゼロ・オフセット・インデクスト

オフセット値が0の場合のインデクスト・アドレシング・モードです．

(例)
```
LDA   0, X
ADDD  0, U
STA   , X
STB   Y
```

オペランド部に書かれたレジスタの内容が実効アドレスを示します．STA，STBのように“0”および“，”は，ほとんどのアセンブラで省略することができます．

(2) コンスタント・オフセット・インデクスト

符号付きの定数をオフセットとするインデクスト・アドレシング・モードです．オフセットの大きさにより，機械語レベルではつぎの3種類に分けられます．

5ビット・オフセット　(−16～+15)

8ビット・オフセット　(−128～+127)

16ビット・オフセット　(−32768～+32767)

命令の語長は，5ビット・オフセットで2バイト，8ビット・オフセットで3バイト，16ビット・オフセットでは4バイトになり，語長が長くなれば，その分だけ実行速度も遅くなります．

通常では，オフセットの大きさによりアセンブラが自動的に上記の3種類を選択するので，特別な場合を除いては，プログラマがこのことを意識する必要はありません．

(例)
```
LDA   8, X
STA   -5, Y
CMPB  $2100, U
```

“，”の前に書かれた部分がオフセット値であり，オフセット値+ポインタ・レジスタの内容が，実効アドレスになります．

図1.4　オート・インクリメント/デクリメント・アドレシングの例

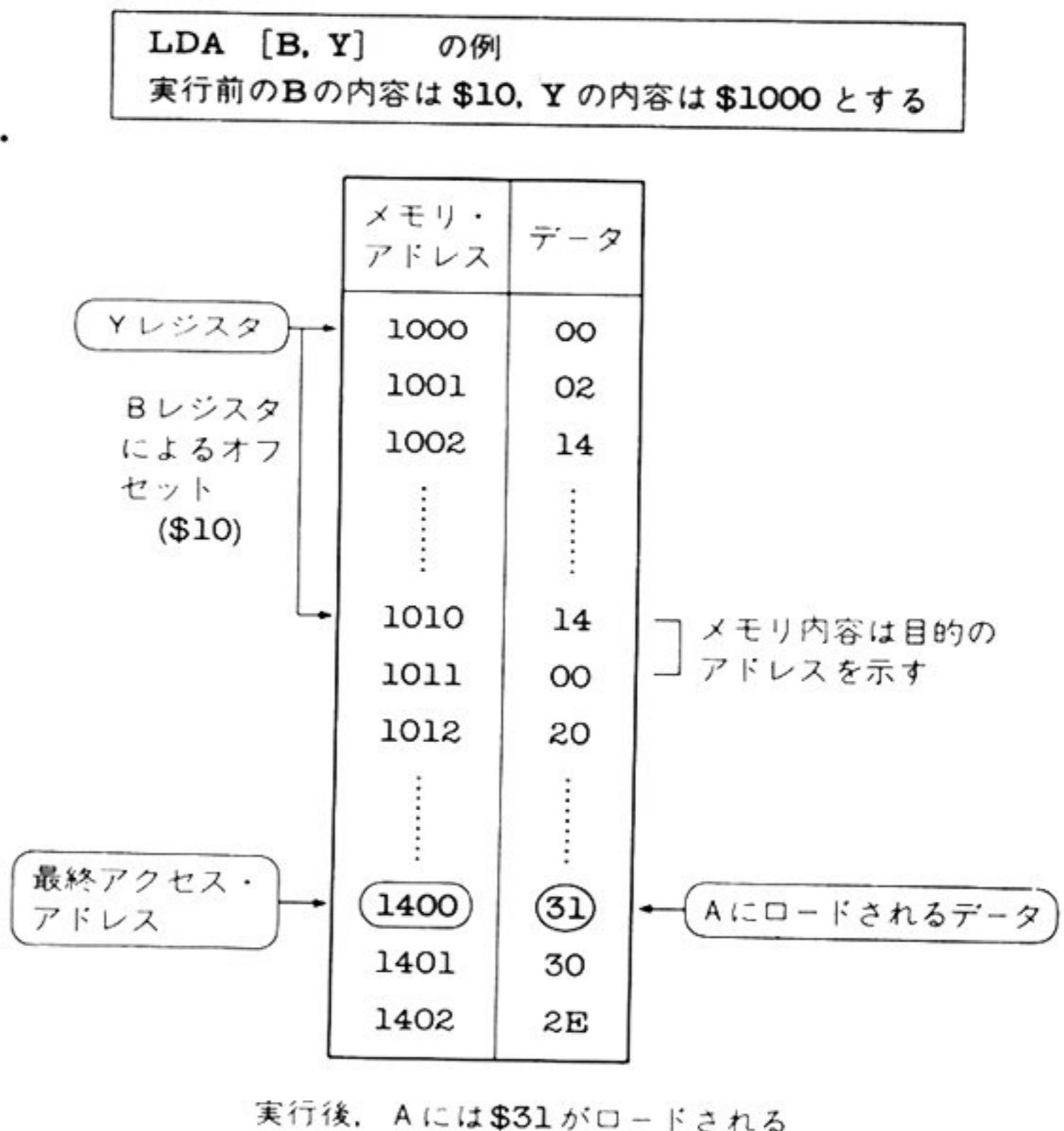
Aには$20がロードされる.
PULS A とほとんど同じだが，コンディション・フラグに与える影響が異なる

Dレジスタの内容が，$1000, $1001にストアされる

(3)アキュムレータ・オフセット・インデクスト

A, BまたはDレジスタ(アキュムレータと呼ばれるレジスタ)の内容をオフセット値とする，インデクスト・アドレシング・モードです.

アキュムレータの内容は符号付き(2の補数表現)2進数として扱われ，実効時にポインタ・レジスタに加算された結果が実効アドレスになります.

```
(例)    LDA     B, X
        LDX     D, Y
        LEAX    A, X
```

(4)オート・インクリメント/デクリメント・インデクスト

オート・インクリメント・インデクスト・モードは，インデクスト・アドレシングが行われた後に使用されたポインタ・レジスタが，自動的に+1または+2されます．これをポスト・インクリメントと呼びます.

オート・デクリメント・インデクスト・モードは，ポインタ・レジスタが−1または−2された後で，インデクスト・アドレシングが行われます．これをプリ・デクリメントと呼びます.

ここで，このモードはスタック・ポインタと同じではないか，と思われる方も多いと思います．図1.4に示すように，LDA ,S+ および PULS A はフラグに与える影響以

図1.5
インデクスト・インダイレクト・アドレシングの例

外は同じ結果になります．すなわち，X, Yはソフト的に，スタック・ポインタとしての使用もできることになります．

参考としてあげれば，68000ではPSH, PULLといった命令はなく，スタック操作はアドレス・レジスタのオート・デクリメント/インクリメントで行います．

```
(例)  LDA   , Y+
      LDD   , X++
      LDA   , S+
      STD   , --Y
```

図1.4を参照してください．

(5) インデクスト・インダイレクト

インデクスト・アドレシングを使用した間接アドレシングです．

+1, −1のオート・インクリメント/デクリメント，および5ビット・オフセットを除くインデクスト・アドレシングでこのモードが使用できます．

図1.6
プログラム・カウンタ・レラティブ・アドレシングの例

このモードでは,インデクスト・アドレスされたメモリの内容はさらにデータのアドレスを示し,すなわち,インデクスによるメモリ間接アドレシングということができます.

```
(例)    STA    [, X]
        LDD    [$44, U]
        LDA    [B, Y]
```

LDAの例を図1.5に示します.

● レラティブ・アドレシング

ブランチ命令はすべてこのアドレシングです.

機械語で見た場合には,オペランド部に符号付き2進数で示されるオフセット値が格納されています.実行時にこのオフセット値とプログラム・カウンタの値が加算され,ブランチ先のアドレス(次に実行するアドレス)が決定します.

オフセットの大きさにより,ショート・レラティブ・アドレシング(1バイト・オフセット)とロング・レラティブ・アドレシング(2バイト・オフセット)があります.

(例)
```
          CMPA  #$41
          BEQ   LB1   (ショート)
          LBRA  JOBA  (ロング)
            :
     LB1  PULS  A, PC
```

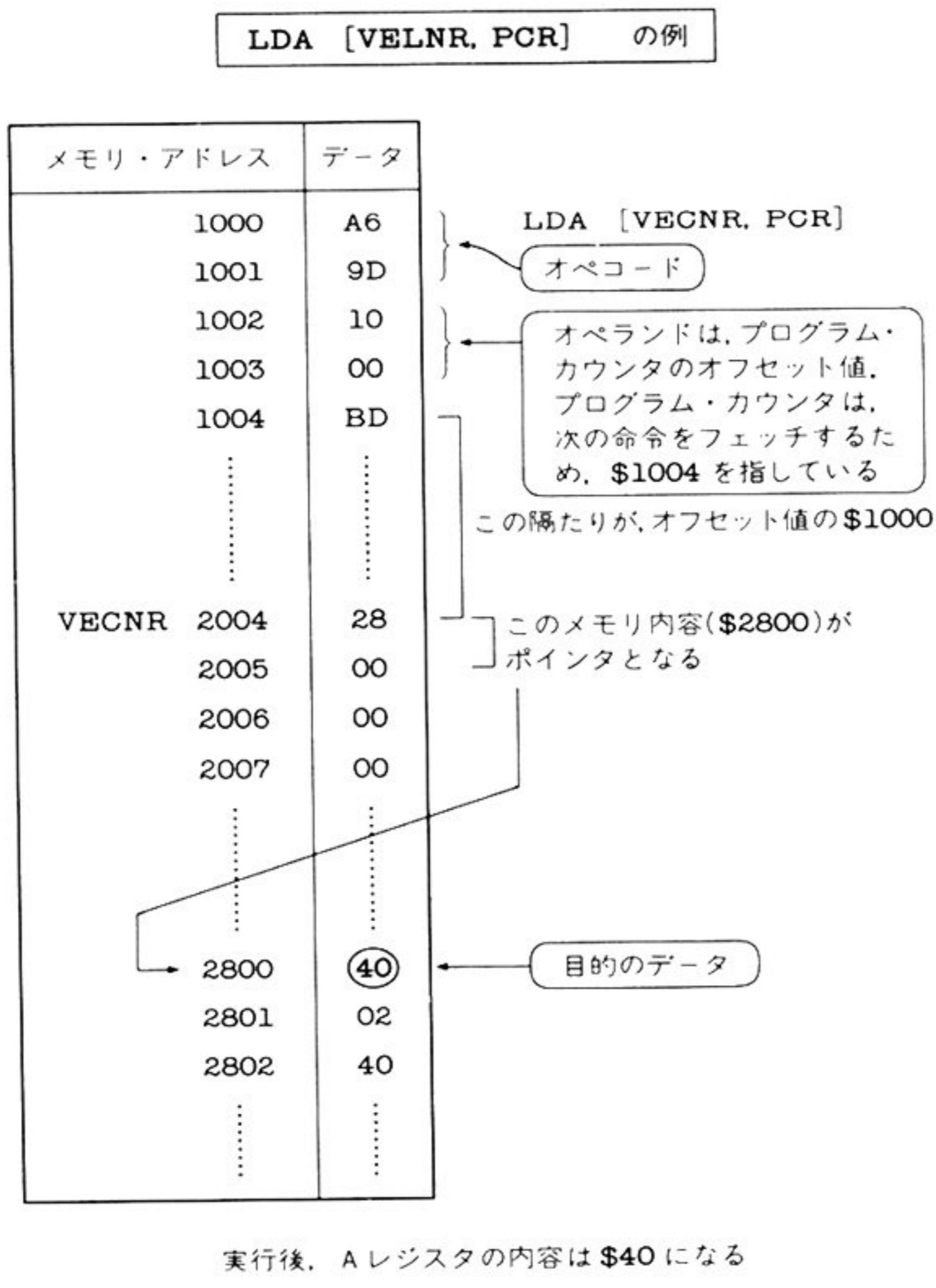
● プログラム・カウンタ・レラティブ・アドレシング

プログラム・カウンタ(PC)をポインタ・レジスタとして，8ビットまたは16ビットの符号付き定数をオフセットとするアドレシング・モードです．

レラティブ・アドレシングでは，次に実行する命令のアドレスを算出するのに対し，ここでは演算の対象となるデータのアドレス算出に使用されます．

このモードは，再配置可能なプログラミングで重要なモードであり，データ・テーブルが自己のプログラム領域内にある場合には，このモードの使用により，アドレスを再配置した場合にも同一のデータを参照することになります．

(例) (図1.6 参照)
```
               LEAX   SINTBL, PCR
               LDB    DIVSER, PCR
                 :
     SINTBL    FDB    175,349,523,698
```

このアドレシングはインデクスト・アドレシングの1形態であるので，インダイレクト・アドレシング，すなわちメモリ間接によるアドレシングも可能です．この場合は，プログラム領域内にアドレス・ポインタとしての変数を設けた場合にも再配置が可能になります．

(例) (図1.7 参照)
```
          LDA    [VECNR, PCR]
```

## 1.3 I/Oのアドレシングについて

Z80を経験し，ここで68系のプロセッサに初めて接する方は，I/Oのアドレシングについてなにも説明されていないことを奇異に感ずるかも知れません．一口にいってしまえば，68系はメモリ・マップトI/Oということなのですが，この言葉が68系の利用者の間で議論

図1.7 プログラム・カウンタ・レラティブ・インダイレクト・アドレシングの例

実行後，Aレジスタの内容は$40になる

されることはあまりなく，Z80の利用者から見た場合にのみ意味をもつ議論のように思います．私たちからしてみれば，当然のこととして受け入れているのです．

メモリ・マップトI/Oとは，I/Oデバイスをメモリと同様に扱います．つまりハード的には，アドレス・デコード回路によってメモリを希望するアドレス空間に割り付けるのと同じ方法で，システム設計者が希望するアドレス領域に割り付けます．ソフト的にも，メモリの読み書きと同様にI/Oの読み書きを行います．

Z80の支持者からすれば，メモリの一部を削ってI/Oのために割り当てなくてはならないので，なにか損をしたように思われるかも知れません．しかし，I/Oのアクセスや演算処

理についても，すべてのアドレシング・モードや処理機能が使用でき，考え方も統一的であるので，はるかに扱いやすいと思うのですが．使い馴れたものはどうしても，ひいきめに見てしまうので，両者を比較する議論はこのくらいで差し控えておきます．

68系のプロセッサ，つまり6800，6809および16ビットの68000も，すべてI/Oに関しては同様に扱います．

### ポジション・インディペンデント

作成された機械語プログラムが，どのアドレスに移されても正しく動作するように作られたプログラムをポジション・インディペンデントと呼びます．

このようなプログラムであれば，機械語のサブルーチン・パッケージを再利用する場合や，システムの都合で最終的な実行とデバッグ時で，配置するアドレスが異なってしまう場合などではたいへん都合がよいわけです．

これを実現するためには，プログラム中で，絶対番地指定を行ってはなりません．つまりエクステンデド・アドレシング・モードは使わないようにすることです．

このためには，プログラム内部の番地を参照するする場合には，プログラム・カウンタ相対アドレシングを使用すればよいわけです．

自身のプログラム内部へのJMPやJSR命令は，BRA，BSR(またはLBRA，LBSR)に置き換えればよく，LDX　#LABEL1のような場合には，

```
LEAX   LABEL1, PCR
```

のようにします．

# 第2章
# 6809のハードウェア

## 2.1 信号と機能の説明

6809のピン配列を図2.1に示します．6800用のペリフェラル・デバイスは，6809でも使用することができます．バス・タイミングもほとんど同様であるので，6800と6809はピン・コンパチブルではないかと，ささやかな期待をもたれる方もあると思いますが，残念ながらピン配列はまったく異なります．

各信号の機能を説明します．

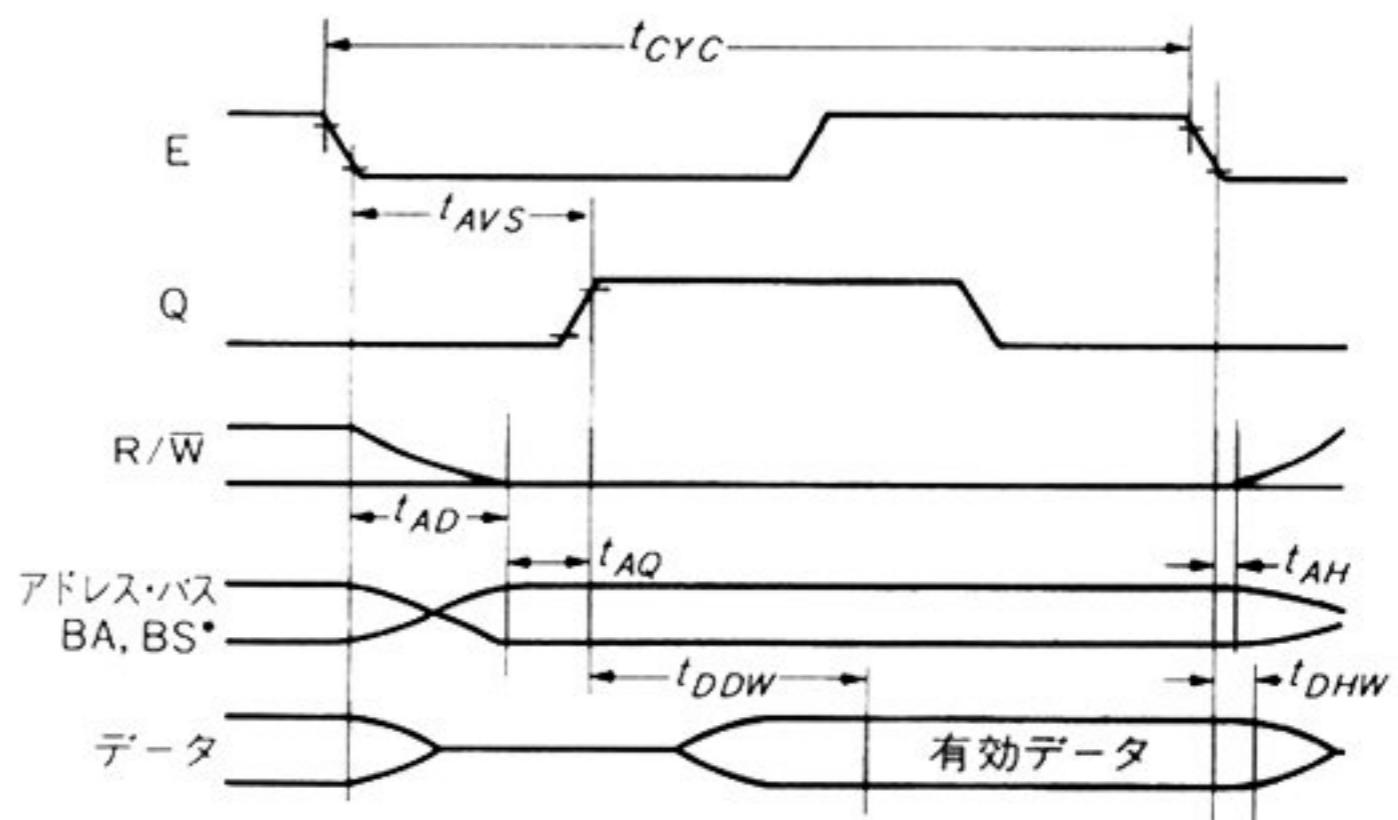
図2.1[2]
6809のピンの配置図

図2.2[2]
6809のリード・タイミング

| | 6809 | | 68B09 | | HD63C09E | |
|---|---|---|---|---|---|---|
| | min | max | min | max | min | max |
| $t_{CYC}$ | 1000 | 10000 | 500 | 10000 | 333 | 2000 |
| $t_{PWEL}$ | 430 | 5000 | 210 | 5000 | 140 | 1000 |
| $t_{AVS}$ | 200 | 250 | 80 | 125 | 65 | — |
| $t_{AQ}$ | 50 | — | 15 | — | — | 110* |
| $t_{AH}$ | 20 | — | 20 | — | 20 | — |
| $t_{DSR}$ | 80 | — | 40 | — | 20 | — |
| $t_{DHR}$ | 10 | — | 10 | — | 20 | — |

* Eクロックの立ち下がりからの時間(アドレス遅延時間)

(単位：ns)

● ($A_0$～$A_{15}$) アドレス・バス

ほかの8ビット・プロセッサもほとんどそうであるように，16ビットから構成され，64Kバイトのアドレス空間を直接アクセスするアドレス情報を出力します．

アドレス・バスの内容はQ信号の立ち上がりまでに確定し，バス・サイクルが終了するまで有効です．MPUの内部処理だけが行われ，アドレス情報が無効の場合は$FFFFが出力されます．

● ($D_0$～$D_7$) データ・バス

データの転送を行う8ビットの双方向性バスです．この信号はE信号に同期して転送が行われますが，タイミングの詳細はリード・タイミング(図2.2)およびライト・タイミング(図2.3)を参照してください．

図2.3[2]
6809のライト・タイミング

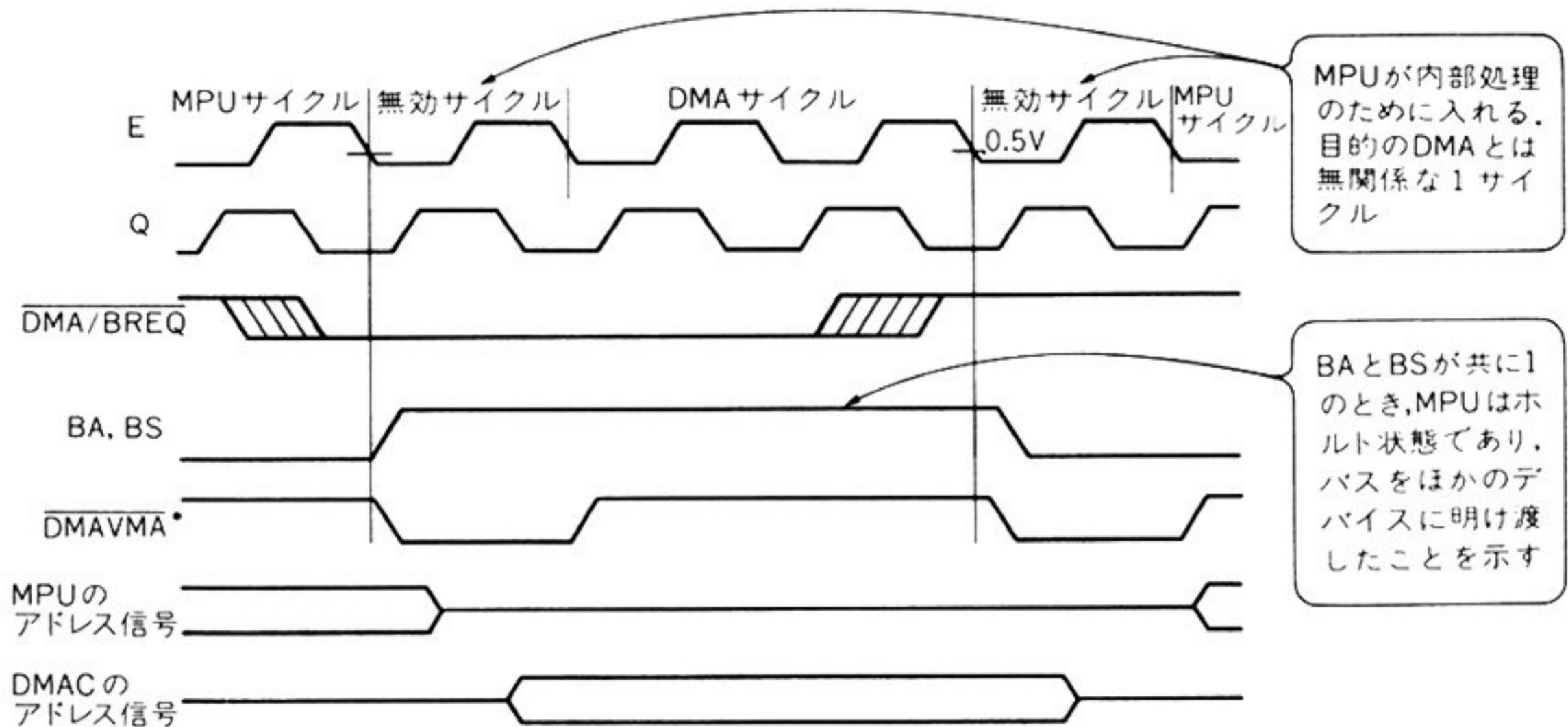
| | 6809 | | 68B09 | | HD63C09E | |
|---|---|---|---|---|---|---|
| | min | max | min | max | min | max |
| $t_{DDW}$ | — | 200 | — | 110 | — | 70 |
| $t_{DHW}$ | 30 | — | 30 | — | 30 | — |

上表以外のタイミングは，**図2.2**の付表を参照　　（単位：ns）

## ● (R/$\overline{\text{W}}$) リード/ライト

データ・バス上のデータの方向を示します．“1”のときはメモリや周辺デバイスからプロセッサへの方向，つまりプロセッサがリードのとき，“0”はMPUからメモリや周辺デバイスへの方向，つまりMPUがライトのときです．

この信号はQ信号の立ち上がりまでに確定し，バス・サイクルの終了まで有効です．

## ● (E)

6800で$\phi$2と呼ばれるクロックに相当します．

このEパルスはバス制御の同期信号として使用され，MPUへのデータはEの立ち下がりで内部に取り込まれます．

## ● (Q)

Q信号は，E信号より1/4クロック進んだタイミングでプロセッサより出力されています．

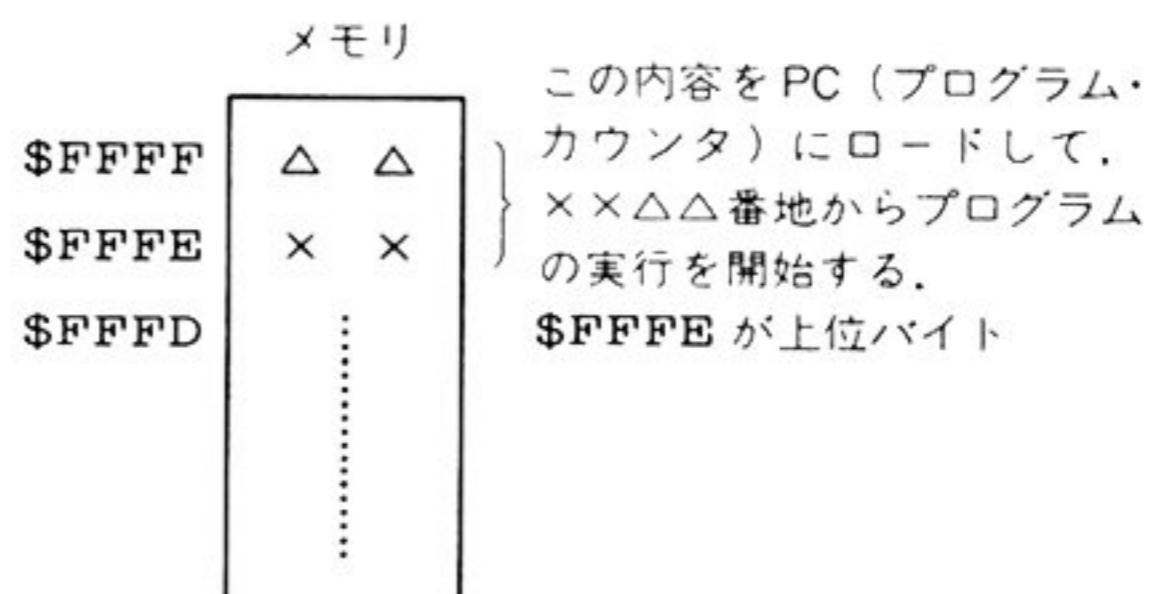
バスの同期信号としてはE信号に並んで重要な信号であり，Qの立ち上がりはアドレス信号の確定を示しています．図2.2と図2.3を参照してください．

図2.4(2) 6809のDMAのタイミング

（＊）$\overline{\text{DMAVMA}}$は，無効サイクル期間でのメモリのアクセスを禁止する必要があれば用いる信号で，外部回路で作る．BAが変化した後の1サイクルを“L”とする．

無効サイクルとは，データ・バスやアドレス・バスの信号が不定であり，意味をもたない状態．無効サイクルの開始と同時に，BAとBSを1にしてホルト状態となったことを示すが，MPUからのアドレス信号はホールド時間があるため，少しの間有効な状態を続ける．このためバスの競合をさけるためにも，無効サイクルでのバスの使用は行うべきでない．バスのほとんどの信号のレベルは常に変化しているため，信号レベルとして意味をもつ状態，すなわち有効と意味をもたない状態，無効との時間があり，ハードウェアの設計ではこのタイム・マージンが重要なポイントとなる．

### ● (MRDY) メモリ・レディ

低速のメモリや周辺デバイスをアクセスするための入力信号です．“1”レベルのときには，EとQは通常の連続したクロックになり，“0”レベルではEとQが1/4マシン・サイクルの単位で引き延ばされます．

引き延ばしの最大時間は15μsです．6809は，ダイナミックRAMと同様に，静止の状態を長く続けることはできません．クロックにも下限があり，100 kHz以上で使用しなければなりません．

### ● ($\overline{\text{DMA}}$/$\overline{\text{BREQ}}$) DMA/バス・リクエスト

プロセッサに対して，命令の実行を一時的に中断し，バスの解放を要求する入力信号です．この用途には，DMA転送やダイナミックRAMのリフレッシュなどがあります．

図2.5[2]
6809のリセット・スタートの様子

この場合に注意を要するのは，この信号を連続して“0”レベル(解放要求)にすることは可能ですが，バスが無制限に使用できるということではないのです．6809はダイナミック・デバイスであるので，14マシン・サイクルごとに内部レジスタのリフレッシュのために，3サイクルの無効期間が発生します．このタイミングはBA，BS信号によって示されます．図2.4を参照してください．

参考に述べておきますが，68000ではバス使用時間の制限はありません．68000では内部の演算処理とインストラクションのフェッチとが，ロジック的にも独立性が高いことによるためであると思います．ただし，68000もダイナミック・デバイスであるので，クロックの下限は定められています．

● ($\overline{\text{RES}}$) リセット

プロセッサをリセット・スタートさせる入力信号です．1マシン・サイクル以上の0レベル入力により，MPUはリセット・シーケンスを開始します．リセット・ベクタは$FFFE，$FFFFであり，この2バイトをフェッチして，その内容が実行開始アドレスになります（図2.5参照）．

この入力には，シュミット・トリガ回路が内蔵されており，*CR*だけの簡単な回路により時間遅れを発生させてリセット信号とすることもできます．

これも参考ですが，68000のリセット・ベクタは最下位の$000000であり，ベクタ・アドレスに書かれたスタック・ポインタの値とプログラム・カウンタの値をロードして実行を開始します．

● (HALT) ホルト

MPUの実行を停止させる入力信号です．この入力により，MPUは実行中の命令が終了した後で停止します．

**表2.1**　バス状態を表すBA, BS信号とMPUの状態

| BA | BS | MPUの状態 |
|---|---|---|
| 0 | 0 | ノーマル |
| 0 | 1 | インタラプト・アクノレッジ |
| 1 | 0 | SYNCアクノレッジ |
| 1 | 1 | HALT |

ノーマル：通常の実行状態

インタラプト・アクノレッジ：
$\overline{RES}$, $\overline{NMI}$, $\overline{FIRQ}$, $\overline{IRQ}$, SWI, SWI1, SWI2の割り込み要求に対して，ベクタ・アドレスを出力している状態

SYNCアクノレッジ：
SYNC命令の実行後，IRQラインからの同期信号を待っている状態

HALT：MPUのホルト状態

ホルト状態では，アドレス・バス，データ・バス，R/W信号はハイ・インピーダンスであり，ほかのデバイスがバスを使用することができます．

このホルト状態には時間の制限はなく，MPUは内部でリフレッシュを続けるので，無制限にMPUの実行を停止させることができます．

ホルト状態では，割り込み要求($\overline{IRQ}$, $\overline{FIRQ}$)に対して応答しませんが，$\overline{NMI}$, $\overline{RES}$の入力に対しては，ホルト解除後の応答に備えてMPU内部にラッチされます．

ホルト状態でも，Q, Eの両信号は通常のクロック出力を続けます．

### ● (BA, BS) バス・アベイラブル，バス・ステータス

BA信号は，アドレス・バス，データ・バス，R/W信号がハイ・インピーダンスになり，ほかのデバイスがバスを使用できることを示す出力信号です．

BS信号はBAとコード化され，MPUの状態を示します．**表2.1**を参照してください．

### ● 割り込み入力

割り込み入力端子には，次に述べる3本があります．それぞれにはベクタ・アドレスが割り当てられており，ほかのベクタも含めて**表2.2**に示しておきます．

#### ▶ ($\overline{NMI}$) ノン・マスカブル・インタラプト

プログラムでマスク不可能な割り込み要求入力であり，$\overline{FIRQ}$, $\overline{IRQ}$よりも高い優先度をもっています．

$\overline{NMI}$が受け付けられると，MPUは内部のレジスタをSスタックに退避し，ベクタで示されるアドレスに実行が移ります．リセット・スタート後はスタック・ポインタSにデータがロードされるまではNMIを受け付けませんが，信号入力はラッチされますので，Sに

表2.2　6809の割り込みベクタ・アドレス

| メモリ・アドレス | | 割り込み |
|---|---|---|
| 上　位 | 下　位 | |
| FFFE | FFFF | $\overline{\text{RES}}$ |
| FFFC | FFFD | $\overline{\text{NMI}}$ |
| FFFA | FFFB | SWI |
| FFF8 | FFF9 | $\overline{\text{IRQ}}$ |
| FFF6 | FFF7 | $\overline{\text{FIRQ}}$ |
| FFF4 | FFF5 | SWI2 |
| FFF2 | FFF3 | SWI3 |
| FFF0 | FFF1 | 予　備 |

予備とは，将来においてMPUの機能を拡張した場合に使用されるかも知れないベクタ・アドレスであり，ユーザはほかの目的にこのアドレスを使用すべきでない

優先度について

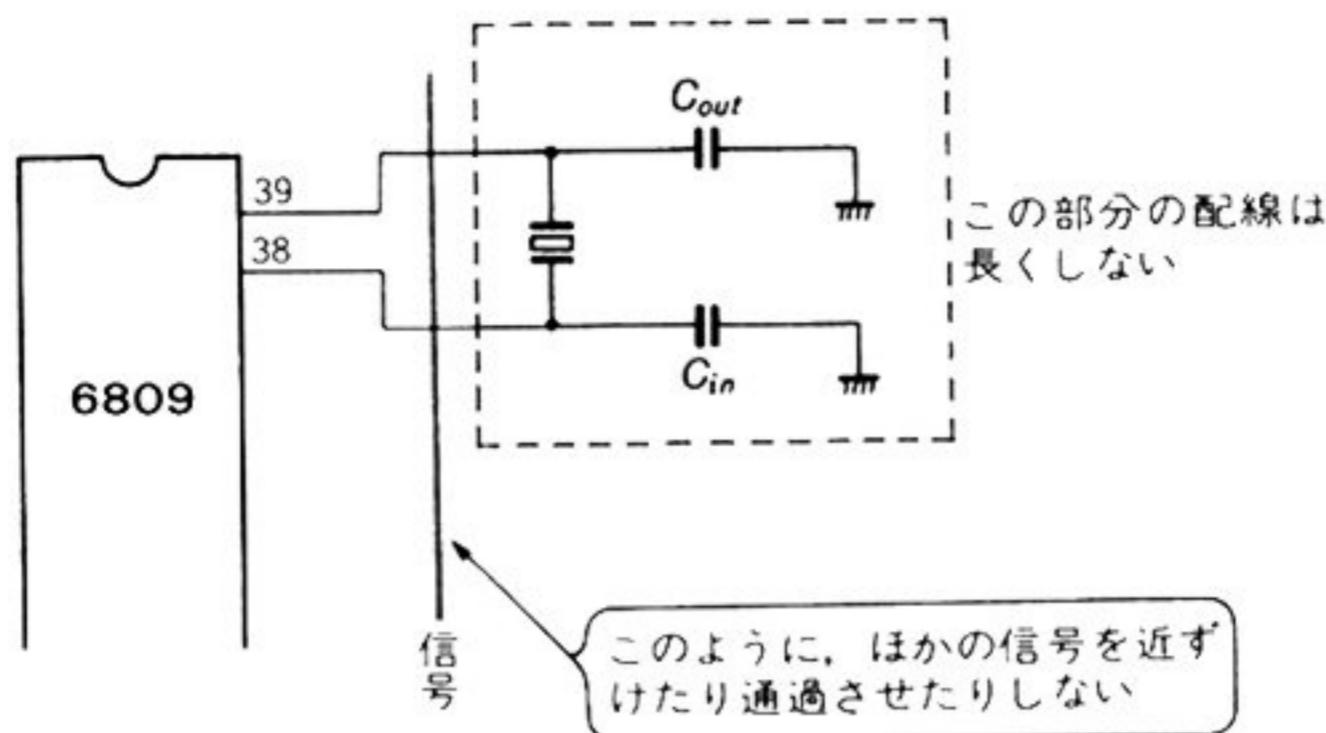
外部の信号入力により割り込みについて優先度の高いものから並べると，$\overline{\text{RES}}$，$\overline{\text{NMI}}$，$\overline{\text{FIRQ}}$，$\overline{\text{IRQ}}$の順になる．
SWI，SWI2，SWI3はソフトウェア割り込みであり，ほかの割り込みと優先度を比較するのは適当ではないが，SWIは$\overline{\text{FIRQ}}$と$\overline{\text{IRQ}}$のマスク・ビットをセットすることから$\overline{\text{FIRQ}}$と同レベルと見ることもできる．
SWI2とSWI3は，ほかの割り込みをマスクすることはなく，最も低い優先度となる．

(注)　$\overline{\text{RES}}$はリセット・スタートであり，割り込みとは別の性格のものだが，プログラム実行の流れに，外部の信号によって直接に影響を与えるということで，同種の機能として説明した．68000では，割り込みもリセットも，どちらも例外処理と呼び，同じカテゴリとして議論される．

データがロードされた直後にNMIシーケンスが起動されます．

▶ ($\overline{\text{FIRQ}}$)　ファースト・インタラプト・リクエスト

6800にはなかった割り込み要求入力です．この割り込みではコンディション・コード・レジスタとプログラム・カウンタの内容しかスタックに退避しないため，たいへん速く割り込みルーチンを起動することができます．

この割り込みは，$\overline{\text{IRQ}}$よりも優先順位が高くなっています．

割り込みサービス・ルーチンでは，RTI(リターン・フロム・インタラプト)命令を実行する前に，割り込み源をクリアしておくことが必要です．

▶ ($\overline{\text{IRQ}}$)　インタラプト・リクエスト

ハードウェアによる割り込み要求としては，最も優先順位の低い入力です．この割り込みでは，Sを除くすべての内部レジスタが自動的にスタックに退避されます．

割り込みサービス・ルーチンではRTIを実行する前に，割り込み源をクリアしておくこ

図2.6　6809水晶発振回路の実装上の注意点

$C_{out}$，$C_{in}$ の容量

| 水晶の周波数 | 4 | 6 | 8 | MHz |
|---|---|---|---|---|
| $C_{in}$，$C_{out}$ | 22 | 20 | 18 | pF |

C-MOSタイプの3MHzバージョンであるHD63C09は，外部クロック・タイプのHD63C09Eのみが現在入手可能

とも忘れないでください.

● (XTAL, EXTAL)

水晶振動子を直接に接続するか，またはクロックの入力端子です.

6809では発振回路が内蔵されており，水晶振動子とコンデンサを接続するだけでクロックを作ることができます．この場合，発振回路は小レベルのアナログ信号を扱うことになるので，実際の組み立てやプリント基板の設計では，それなりの注意が必要です．図2.6を参照してください.

この端子には，外部で作られたクロックをTTLレベルで入力することもできます．この場合は，クロック信号をEXTAL(38ピン)に入力して，XTAL(39ピン)はグラウンドに落とします.

水晶振動子または外部クロックは，マシン・サイクルの4倍の周波数のものを使用します.

## 2.2　6809バスと68000バスとの関係

6809を使用しているかまたは計画中の方で，次のステップとして68000を考えている方も多いと思います．ここでは，6809の周辺デバイスまたは周辺回路の基板について，68000での利用の可能性についてお話ししておきます．

### 割り込みベクタ・アドレスの書き換え

割り込みベクタはの表2.2に示したように，アドレス空間の最上位の部分に位置しています．

リセット・スタートのベクタもこの領域に含まれているので，この領域はモニタ・プログラムの一部としてROMが配置されている場合が多いのです．従って，これらのベクタ・アドレスはモニタの管理下に置かれるのが普通であり，内容の変更はモニタの仕様にしたがって行います．

この変更は，モニタがベクタ・テーブルとして定めたRAM領域をユーザ・プログラムによって書き換える場合と，モニタのサービス・ルーチンを利用する場合があり，ASSIST09では，後者の方法で行います．いずれにしてもモニタの説明または仕様を参照して，それにしたがってください．

ASSIST09で，プリンタ・スプーラの$\overline{\mathrm{IRQ}}$ベクタを登録する場合の例を以下に示しておきます(第6章参照)．

```
LDX    #PRTSTR
LDA    #12
SWI
FCB    9
```

例のように，Xレジスタには新しくするアドレス値をロードし，Aレジスタには，ASSIST09のベクタ・テーブルでのIRQを示すコードである12をロードし，SWI命令でサービス・ルーチンをコールします．FCB 9は，ベクタ・スワップを示すエントリ・コードです．

6809では(8ビット・プロセッサのほとんどがそうですが)，データ転送の基本はMPUから出力されるバス・クロックに従って行われます．つまり，同期式バスと呼ばれるものです．一方，68000では非同期バスが基本であり，MPUは周辺に対してストローブ信号を送り，周辺はアクノリッジ信号を返す，といったタイミングでバス上のデータ転送が行われます．

この詳細については，68000関連マニュアル(たとえばトランジスタ技術スペシャル〈2〉，基礎から学ぶMC68000)を参照するようお願いしますが，この両者のバス・タイミングはまったく異なるものです．そのため，6809の周辺を68000で利用することは困難のように思われるかも知れませんが，68000のほうでこの点についての用意がなされています．68000は，一時的にバス・タイミングを6800に相当するタイミングでバスのアクセスを行うことができるのです．

実際の68000システムにおいても，相当量の6800や6809の周辺デバイスが使用されています．初期のシステムにおいては，ポートやタイマなどのすべての周辺デバイスが8ビット用のものである，といったシステムが立派な68000マシンとして稼動しているのです．

# 第3章
# CPUボードの設計例

## 3.1 CPUボードの回路

6809を使用したシステムやCPUボードの例は，これまでにも数多くが発表されていますが，ここでは工業計測や自動制御を主な目的として設計され，現在も新たな目的の中で使い続けられている例を紹介します．

図3.1にCPUボードの回路図，**写真3.1**にその外観を示しますが，計測や制御用といっても，一部にその考慮がされているだけであり，特別なものではありません．これだけでワン・ボード・コンピュータとして必要な機能は一通り備えており，筆者の場合では，これに64KのDRAMとFDC(フロッピ・ドライブ・コントローラ)の乗ったボードを組み合わせて，プログラム開発ツールとして使っています．

計測制御用ということで考慮した特徴について，以下に列記しておきます．

▶プログラムはROM化して使用することがほとんどであるため，ROMは64Kバイトまで実装可能とした(RAMも必要なので，実際には64KすべてがROMということはないが)．

▶リアル・タイムな高速処理を考慮して，割り込みが拡張されている．

IRQ入力は一本でなく，優先順位をもった8本に拡張されています．

▶マルチ・タスク・モニタを使用することにも対処した．

▶タイマ(6840)をボード内に実装可能とし，時間測定に備えた．

▶多量のインターフェース基板を伴うことを予想して，周辺デバイスのアドレスとROM，RAMのアドレスの一部が同一のアドレスになった場合，バスの衝突を避けるため，優先

図3.1(a)　6809 CPU ボード回路図

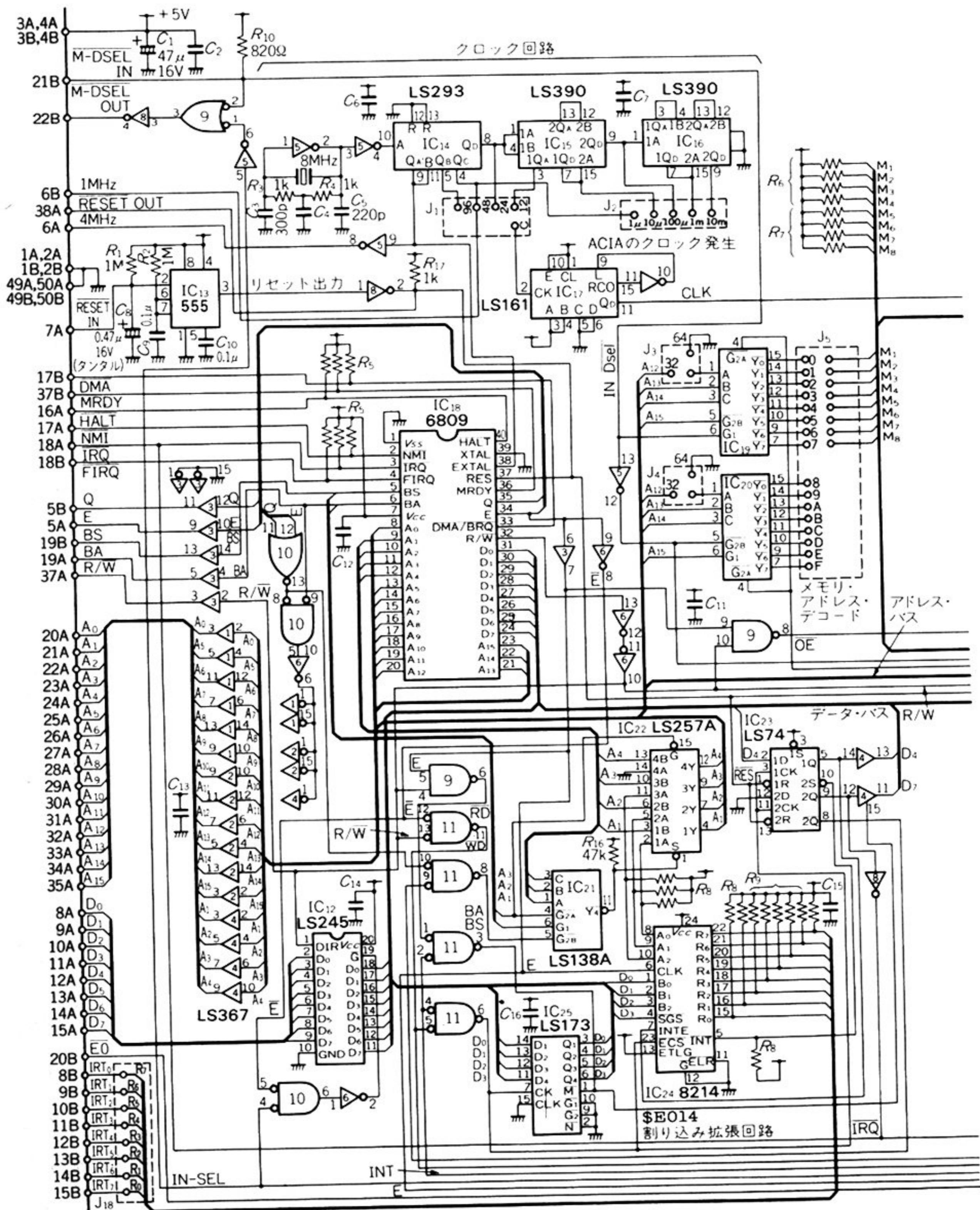



図3.1(b) 6809 CPU ボード回路図（つづき）

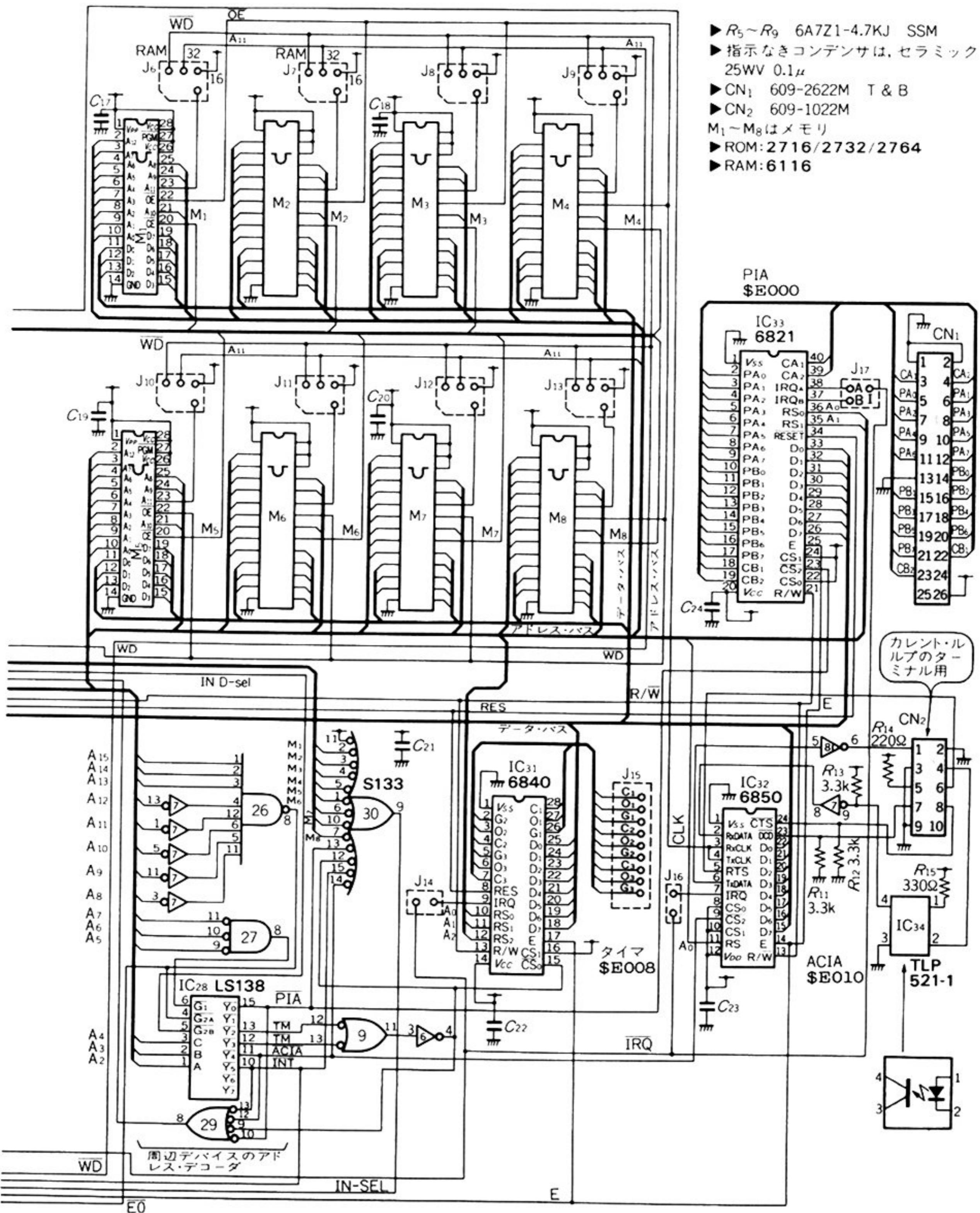
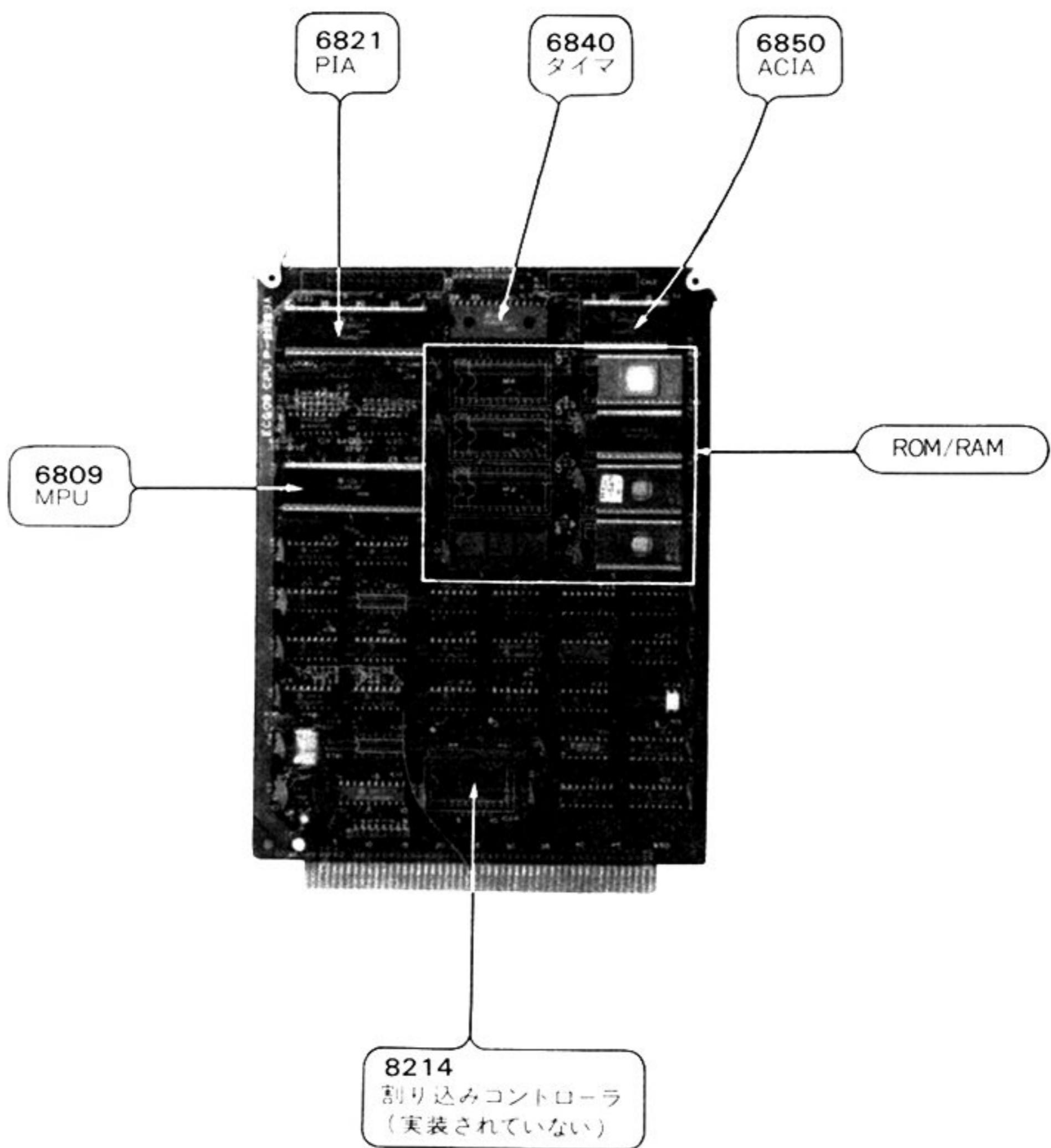



図3.1(c) CPUボードの部品表

| IC 番号 | デバイス名 | ピン数 | GND | +5V | 備考 |
|---|---|---|---|---|---|
| $IC_{1,2,3,4}$ | 74LS367 | 16 | 8 | 16 | |
| $IC_{5,6}$ | 74LS04 | 14 | 7 | 14 | |
| $IC_7$ | 74LS14 | 14 | 7 | 14 | |
| $IC_8$ | 7406 | 14 | 7 | 14 | |
| $IC_9$ | 74LS00 | 14 | 7 | 14 | |
| $IC_{10}$ | 74LS02 | 14 | 7 | 14 | |
| $IC_{11}$ | 74LS32 | 14 | 7 | 14 | |
| $IC_{13}$ | 555 | 8 | | | |
| $IC_{14}$ | 74LS293 | 14 | 7 | 14 | |
| $IC_{15,16}$ | 74LS390 | 16 | 8 | 16 | |
| $IC_{17}$ | 74LS161A | 16 | 8 | 16 | |
| $IC_{18}$ | 6809 | 40 | | | モトローラ，日立 |
| $IC_{19,20,21,28}$ | 74LS138 | 16 | 8 | 16 | |
| $IC_{22}$ | 74LS257A | 16 | 8 | 16 | |
| $IC_{23}$ | 74LS74A | 14 | 7 | 14 | |
| $IC_{24}$ | 8214 | 24 | 12 | 24 | インテル |
| $IC_{25}$ | 74LS173 | 16 | 8 | 16 | |
| $IC_{26}$ | 74LS30 | 14 | 7 | 14 | |
| $IC_{27}$ | 74LS27 | 14 | 7 | 14 | |
| $IC_{29}$ | 74LS20 | 14 | 7 | 14 | |
| $IC_{30}$ | 74S133 | 16 | 8 | 16 | |
| $IC_{31}$ | 6840 | 28 | 1 | 14 | モトローラ，AMI |
| $IC_{32}$ | 6850 | 24 | 1 | 12 | モトローラ，日立 |
| $IC_{33}$ | 6821 | 40 | 1 | 20 | |
| $IC_{34}$ | TLP521-1 | 4 | | | 東芝 |
| $M_1 \sim M_8$ | 2764 | 28 | 14 | 28 | 日立，TI |
| | 2732<br>or 6116 | 24<br>24 | 12<br>12 | 24<br>24 | |

順位を自動的に設定する工夫が成されている．

以上のようなわけで，十分な実績もあり，かなり満足している基板ではあるのですが，問題がないわけではありません．このCPUボードを設計してから，数年が経過しているため，対応しているメモリが現在では古く感じます．27128(16Kバイト EP-ROM)や6264(8Kバイト RAM)も使用できるようにすべきです．ターミナル・インターフェースも，筆者が古くからの習慣を引摺ってきているため，簡易のカレント・ループが使われていますが，RS-232Cのほうが一般的です．これらの問題については，後で触れることにします．

写真3.1 CPUボードの外観

6821
PIA
6840
タイマ
6850
ACIA
6809
MPU
ROM/RAM
8214
割り込みコントローラ
(実装されていない)

以下，回路の要点について説明します．

● リセット回路

555($IC_{13}$)により，電源投入時または，リセット・スイッチが押された場合に，ワン・ショット・パルスを発生させています．

● クロック回路

6809は水晶振動子を直接MPUに接続することもできますが，MPUやバス・クロック以外のクロックも同時に得るため，MPUの外部にクロック回路を用意しました．

ACIA(シリアル・ポート)のクロックは，発振された8MHzをバイナリ・カウンタ($IC_{14}$)

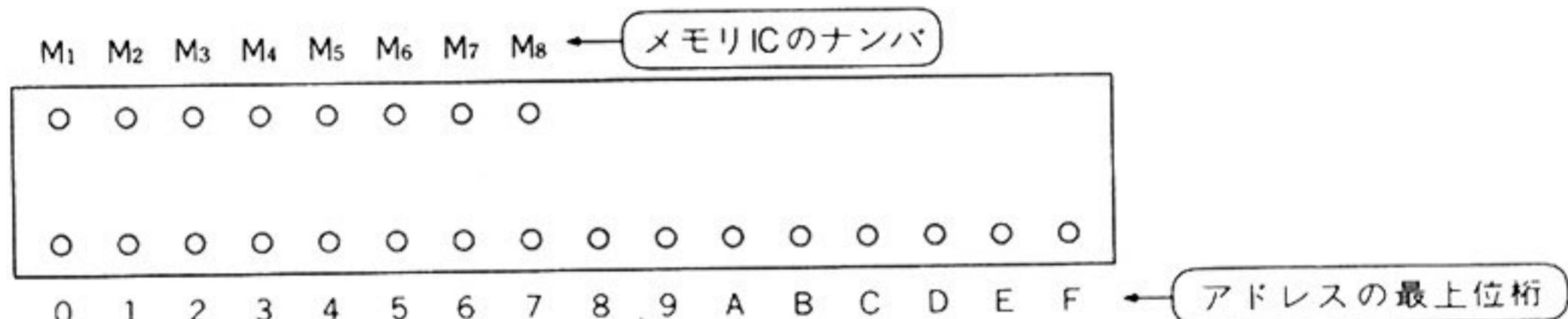
表3.1　CPUボードのジャンパ・セレクトの設定例

▶$J_6$～$J_{13}$　メモリICのセレクト

RAM　32　16
○　○　○

○
C

2716：Cと16をジャンプ
2732：Cと32をジャンプ
2764：Cと32をジャンプ
6116：CとRAMをジャンプ

▶$J_1$　ボーレイトの選択

96　48　24　12
○　○　○　○

○
C

1200ボー：Cと12をジャンプ
2400ボー：Cと24をジャンプ
4800ボー：Cと48をジャンプ
9600ボー：Cと96をジャンプ

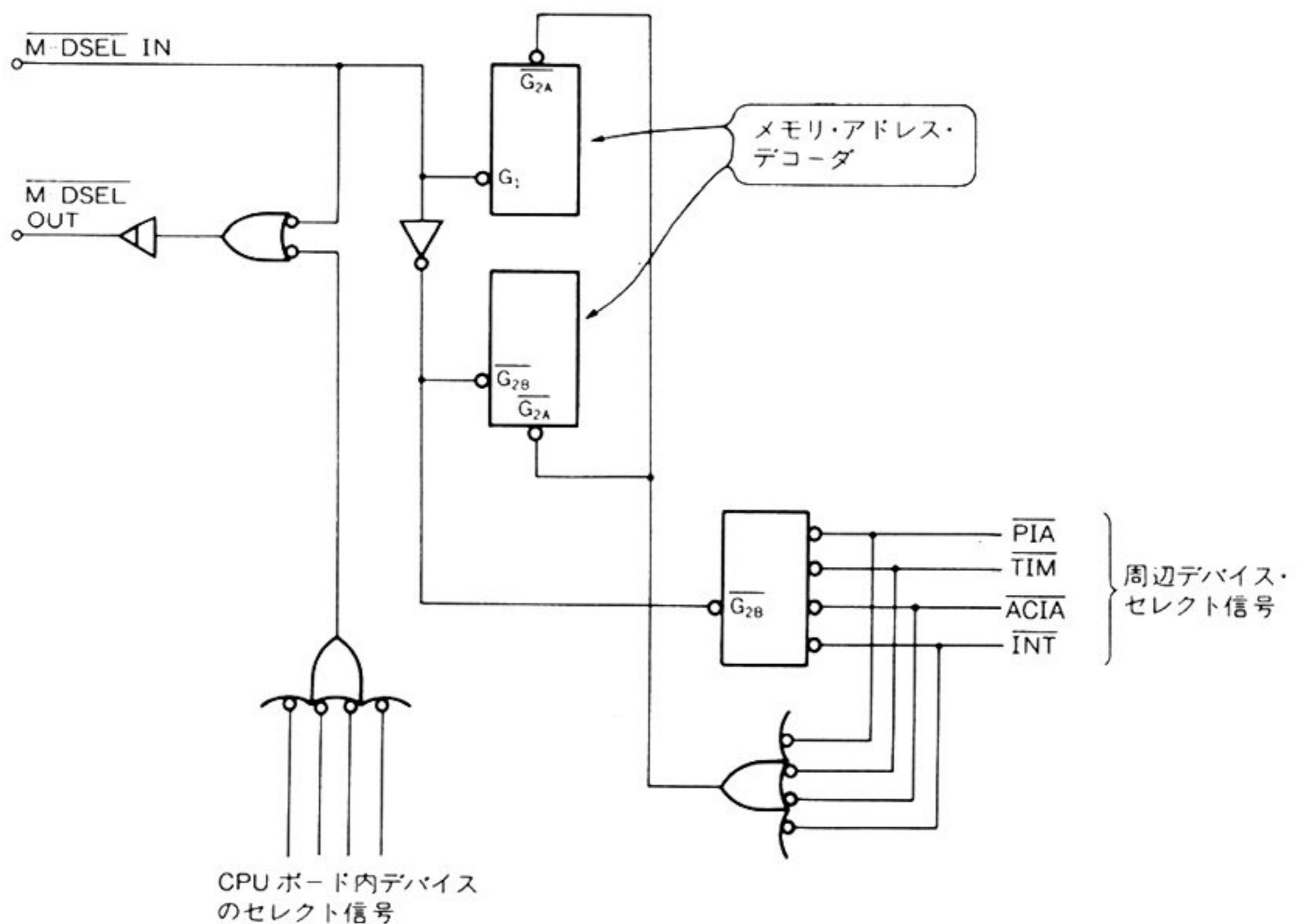
▶$J_3$，$J_4$　マッピングにおけるブロック・サイズ

Cと64をジャンプすれば，メモリICは8Kバイトとみなす．
Cと32をジャンプすれば，メモリICは4Kバイトみなす．
64を選択した場合は，アドレス最上位桁が，偶数のみ有効となる．

▶$J_5$　メモリICのマッピング

ICナンバとマッピングしたい最上位桁とをジャンプする．

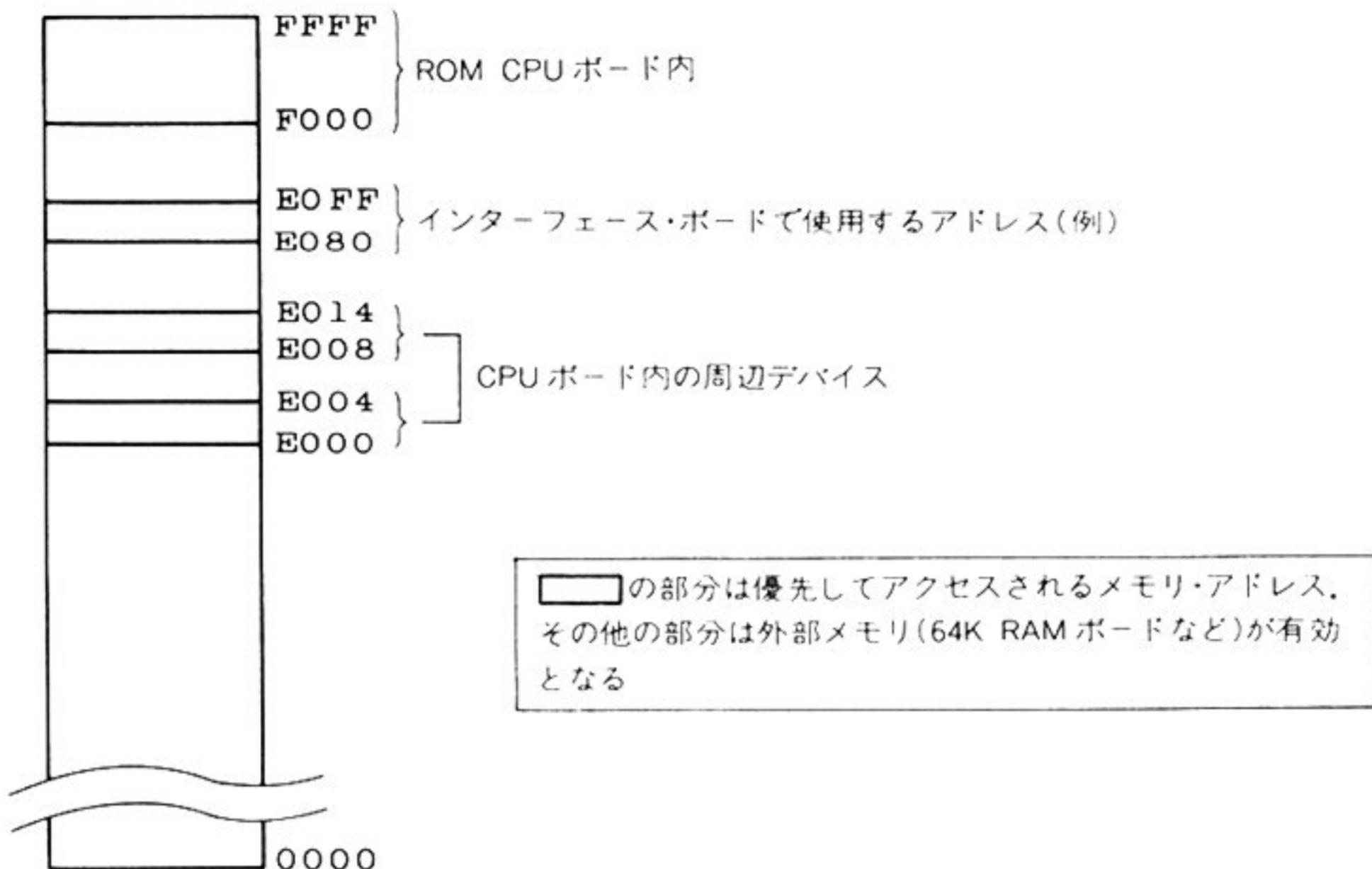
で分周されたものを，さらに$IC_{17}$によって1/13分周して得ています．$J_1$はボーレイトのジャンパ・セレクトです．詳しくは**表3.1**を参照してください．

このクロック回路では，さらに10進カウンタによって最長10 msのクロックを作っていますが，これは後で述べるマルチ・タスク・モニタのクロックとして使用します．

## ● アドレス・デコーダ

メモリ・アドレス・デコーダと周辺デバイスのアドレス・デコーダの二つの部分から構成されています．アドレス・デコーダとしてとくに変わったものではありませんが，同一アドレスに配置されたデバイスの優先順位分けが行われます．

この点について，抜き出した回路を**図3.2**に示します．

$\overline{\text{M-DSEL}}$ INは，CPUボードの外部でデコードされたセレクト信号をそのまま入力し

図3.2 CPUボードのアドレス・デコーダ回路

ます．この信号によって$IC_{19}$, $IC_{20}$, $IC_{28}$のアドレス・デコーダはすべてインヒビットされます．CPUボード内の周辺デバイスのセレクト信号は，$IC_{29}$でORが取られ，メモリ・アドレス・デコーダをインヒビットします．CPUボード内のすべてのセレクト信号は$IC_{30}$でORが取られ，さらに$\overline{\text{M-DSEL}}$ INとORが取られて，$\overline{\text{M-DSEL}}$ OUTとしてCPUボードから出力されます．

このことは，複数のデバイスがメモリ空間の一部を共有する形でマッピングされた場合は，その共有されたアドレスで有効となるデバイスの優先順位はすでに定まっているということです．

優先順位の高いものから並べると，次の順になります．

(1) CPUボード外の周辺デバイス

(2) CPUボード内の周辺デバイス

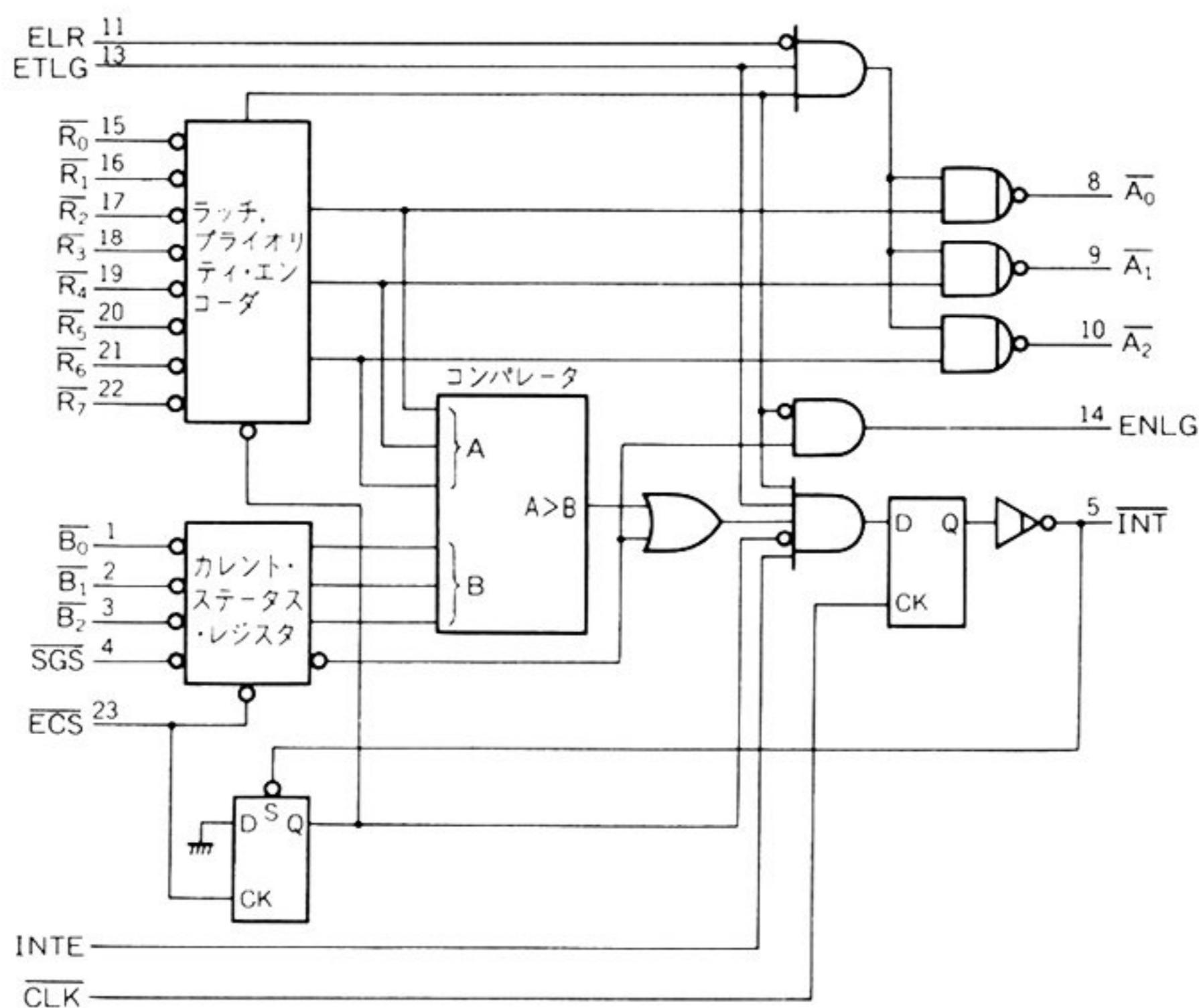
(3) CPUボード内のROM/RAM

(4) CPUボード外のメモリ・デバイス

図3.3　各デバイスのマッピングの例

マッピングの例を図3.3に示します.

この例では，バスに64Kバイトの RAM が実装されている場合ですが，□の部分では上記に示す優先順位の高いデバイスが RAM に代って有効になります.

● **割り込みの拡張回路**

$\overline{\mathrm{IRQ}}$ 入力の拡張と同時に，$\overline{\mathrm{IRQ}}$ ベクタの拡張も行います.

この回路ではインテル社の少々古い IC ですが，割り込みコントローラの 8214 を使用しています．8214 の内部ブロック図を図3.4に示しておきます.

この回路を使用する場合は，バスの $\overline{\mathrm{IRQ}}$ は使用せず，$\mathrm{IRT_0}$～$\mathrm{IRT_7}$を使用します．この割り込み入力は，優先順位が設けられており，$\mathrm{IRT_7}$が最も高い優先レベルになっています.

割り込みが受け付けられると，この回路から MPU に対して $\overline{\mathrm{IRQ}}$ 信号が送られ，MPU が $\overline{\mathrm{IRQ}}$ に応答すれば，$\overline{\mathrm{IRQ}}$ のベクタ・アドレスを出力します．このとき，アドレス信号の下位4ビットを拡張されたベクタ・アドレスに変換して出力します.

拡張された $\overline{\mathrm{IRQ}}$($\mathrm{IRT_0}$～$\mathrm{IRT_7}$)とベクタ・アドレスの関係を，表3.2にまとめておきます.

図3.4[8] 割り込みコントローラ 8214の内部ブロック図

ピン名と機能

| 入　力 | |
|---|---|
| $\overline{R_0}$～$\overline{R_7}$ | 割り込み要求，$R_7$が最も高レベル |
| $\overline{B_0}$～$\overline{B_2}$ | レベル・ステータス |
| $\overline{SGS}$ | ステータス・グループ・セレクト |
| $\overline{ECS}$ | カレント・ステータスを有効 |
| INTE | 割り込み有効 |
| $\overline{CLK}$ | クロック |
| $\overline{ELR}$ | レベル出力を有効 |
| ETLG | このグループを有効 |
| **出　力** | |
| $\overline{A_0}$～$\overline{A_2}$ | 要求レベルの出力 |
| $\overline{INT}$ | 割り込み要求 |
| ENLG | 次のレベルのグループを有効 |

**レベル・ステータス**

この信号には，カレント・ステータス・レジスタに書き込むデータを入力する．MPUとのインターフェースでは，MPUのデータ・バスの下位3ビットに対応する．
これにより，カレント・ステータス・レジスタに書き込まれた内容よりも高いレベルの割り込み要求が受け付けられることになる．

回路についての説明を加えておきます．

$IC_{21}$は，MPUが$\overline{IRQ}$のベクタ・アドレスを出力しているかどうかをモニタします．このためにはBA, BS, $A_1$, $A_2$, $A_3$をデコードすることで知ることができます．

表3.2 拡張された$\overline{IRQ}$ ($IRT_0$~$IRT_7$) とベクタ・アドレス

| 割り込み入力 | ベクタ・アドレス |
|---|---|
| $IRT_0$ | FFE0 |
| $IRT_1$ | FFE2 |
| $IRT_2$ | FFE4 |
| $IRT_3$ | FFE6 |
| $IRT_4$ | FFE8 |
| $IRT_5$ | FFEA |
| $IRT_6$ | FFEC |
| $IRT_7$ | FFEE |

$IC_{22}$は，ベクタ・アドレスの変換を行います．通常は$A_1$~$A_4$をそのまま素通りさせるだけですが，$IC_{21}$がバス上に$\overline{IRQ}$のベクタ・アドレスが発生したことを検出すると，$A_1$~$A_4$は$IC_{24}$が出力する割り込みレベルに変換して出力します．この結果は表3.2に示すとおりです．

$IC_{23}$の半分は，$E014に割り付けた1ビットのレジスタとして動作します．このアドレスの$D_4$を1にすることで，レジスタの出力は$IC_{24}$を割り込みイネーブルの状態にします．残りの半分は，$IC_{24}$から出力される割り込み要求($\overline{INT}$)をラッチして，MPUに$\overline{IRQ}$信号を送ります．

このラッチのクリアは，$E014をリードすることで行えます．$IC_{24}$の動作については，図3.4のブロック図で理解できるものと思います．

$IC_{25}$は，$IC_{24}$に書き込んだ割り込みレベルのステータスを再び読み取るためのレジスタです．このアドレスも$E014です．つまりこの回路全体は，$E014の下位5ビットでコントロールしていることに注意してください．

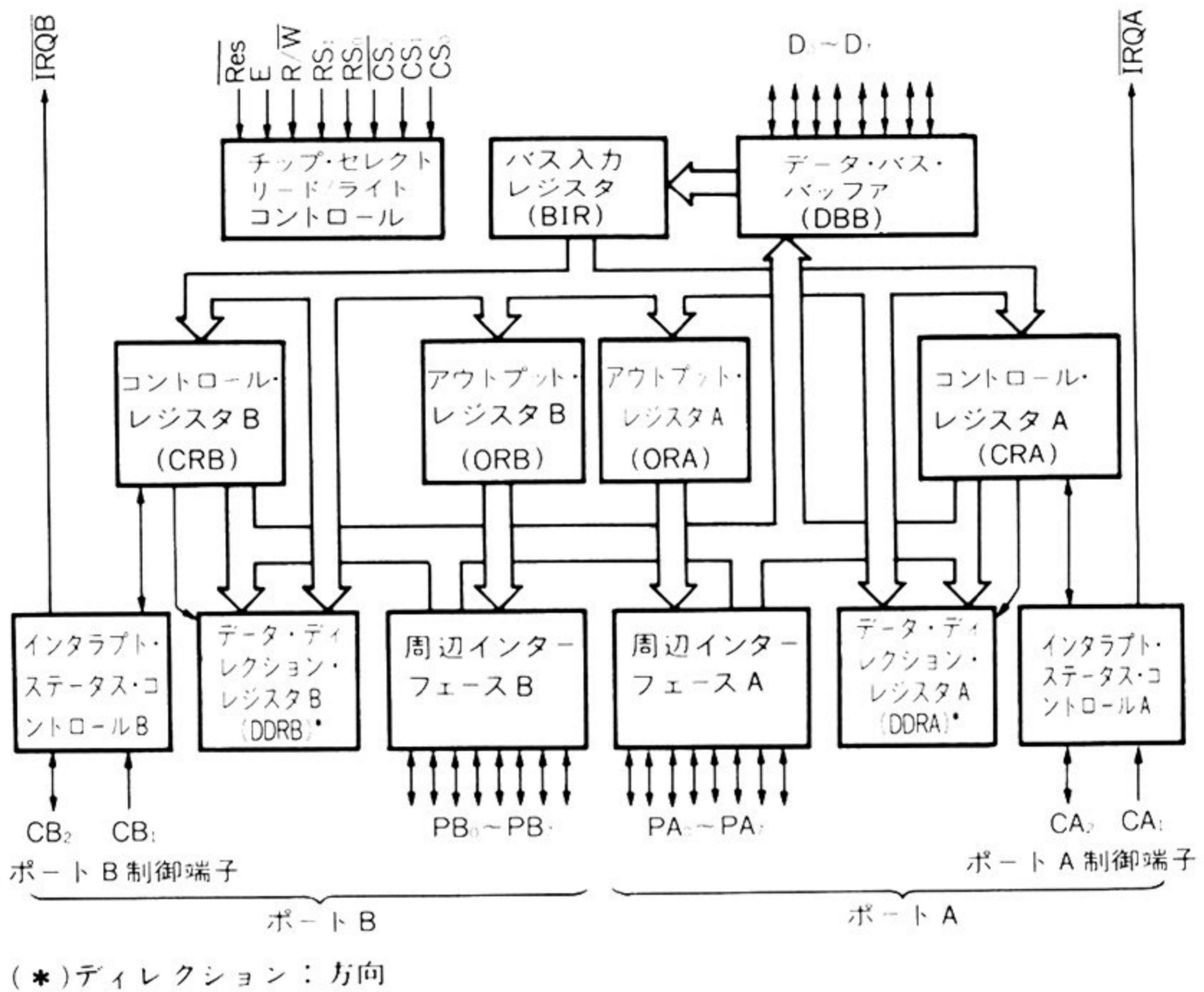
## 3.2 CPUボード内のペリフェラル

紹介したCPUボードで使用している，6809の周辺デバイスについて説明しておきます．これらのデバイスを解説した書籍はたくさんあり，多くの読者にとっては重複かと思いますが，できるだけまとまった6809の資料にしておきたいということもあり，次に述べておきます．

図3.5[2] パラレル・インターフェース6821のピン配置図

| | ピン | | | ピン | |
|---|---|---|---|---|---|
| | 1 | $V_{SS}$ | $CA_1$ | 40 | 入力ポート |
| Aポート | 2 | $PA_0$ | $CA_2$ | 39 | 入力ポート |
| Aポート | 3 | $PA_1$ | $\overline{IRQA}$ | 38 | |
| Aポート | 4 | $PA_2$ | $\overline{IRQB}$ | 37 | |
| Aポート | 5 | $PA_3$ | $RS_0$ | 36 | |
| Aポート | 6 | $PA_4$ | $RS_1$ | 35 | |
| Aポート | 7 | $PA_5$ | $\overline{Reset}$ | 34 | リセット |
| Aポート | 8 | $PA_6$ | $D_0$ | 33 | データ・バス |
| Aポート | 9 | $PA_7$ | $D_1$ | 32 | データ・バス |
| Bポート | 10 | $PB_0$ | $D_2$ | 31 | データ・バス |
| Bポート | 11 | $PB_1$ | $D_3$ | 30 | データ・バス |
| Bポート | 12 | $PB_2$ | $D_4$ | 29 | データ・バス |
| Bポート | 13 | $PB_3$ | $D_5$ | 28 | データ・バス |
| Bポート | 14 | $PB_4$ | $D_6$ | 27 | データ・バス |
| Bポート | 15 | $PB_5$ | $D_7$ | 26 | データ・バス |
| Bポート | 16 | $PB_6$ | E | 25 | 同期信号 |
| Bポート | 17 | $PB_7$ | $CS_1$ | 24 | チップ・セレクト |
| | 18 | $CB_1$ | $\overline{CS_2}$ | 23 | チップ・セレクト |
| | 19 | $CB_2$ | $CS_0$ | 22 | チップ・セレクト |
| | 20 | $V_{CC}$ | $R/\overline{W}$ | 21 | リード/ライト |

### ◆ 6821

PIA(Peripheral Interface Adapter)と呼ばれ，一般的に使われるパラレル入出力ポートです．ピン配置を図3.5，内部ブロック・ダイヤグラムを図3.6に示します．

このポートは8ビットの入出力を2組備えており，それぞれにハンドシェイク，または割り込み入力用のコントロール入出力を2ビットずつもっています．

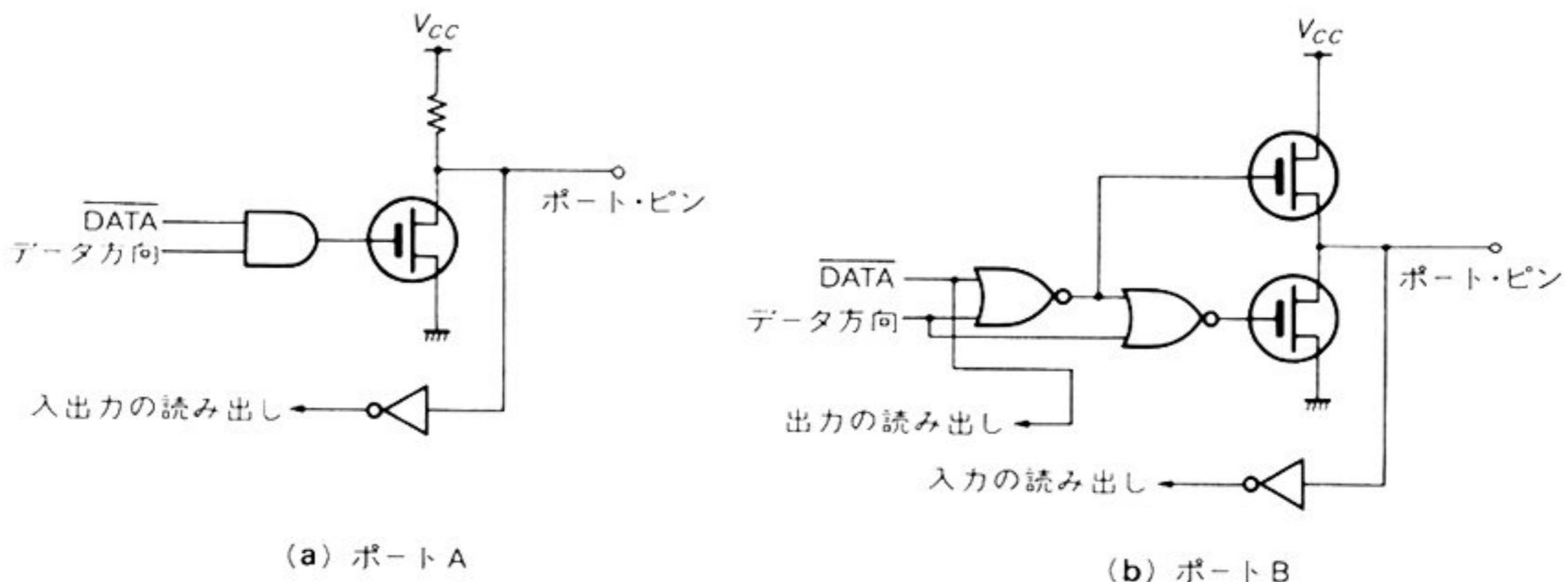
ブロック・ダイヤグラムに示すように，入出力インターフェースは，8ビットの単位でA，Bと呼ばれ，これをAポート，Bポートと呼びます．どちらのポートも1ビット単位で入力または出力，つまりデータの方向をプログラムで設定できます．

AとBは，ほぼ同等の機能をもっていますが，入出力内部の回路構成が異なり，使用目的によっては，使い分ける必要のある場合もあります．等価回路を図3.7に示します．

コントロール信号についても一部に違いがあります．詳細については，表3.3(6821の制御)を参照してください．

表3.3についての注釈を以下に述べておきます．

図3.6[2] パラレル・インターフェース6821の内部ブロック・ダイヤグラム

(*)ディレクション：方向

● テーブル1

このテーブルは，6821の内部レジスタの選択を示しています．$RS_0$, $RS_1$は，通常アドレス・バスの$A_0$, $A_1$に接続され，アドレスの下位2ビットで内部レジスタの選択を行います．さらに，コントロール・レジスタのビット2 ($CRA_2$, $CRB_2$)もレジスタ選択の一部を受けもち，方向レジスタと周辺レジスタの区別を行っていることに注意してください．

周辺レジスタに書き込まれた内容は，ポート出力にそのまま反映されます．この内容は再び読み出すことができます．

方向レジスタは，ポートの入出力方向を決定します．周辺レジスタに対応したビットがそれぞれの方向を決定し，0では入力，1で出力を指定します．

コントロール・レジスタは，各ビットが異なった機能をもちますが，テーブル2以降で詳細を示してあります．

図3.7[2] パラレル・インターフェース6821のポートA，ポートBの等価回路

(a) ポートA　(b) ポートB

ポートAとポートBは上図のように異なるが，意識して使い分けしなくてはならない場合はそれほど多くない．
Aではオープン・コレクタのような信号をプルアップなしでも入力できるが，Bの入出力ではハイ・インピーダンスとなるのでプルアップが必要．出力したデータを読み出す場合もAとBとでは異なる．Aはポート・ピンのレベルを読むが，Bでは内部ロジックで読むので，ポート・ピンの実際のレベルは関係しない．本当に出力したレベルどおりにピンのレベルがなっているかを確認するには，Aを使う必要があるが，このような必要性はあまり生じさせないほうがよい．

表3.3[2] パラレル・インターフェース6821の制御信号テーブル

● テーブル1：内部レジスタの選択

| $RS_0$ | $RS_1$ | コントロール・レジスタ・ビット | | 選　　択 |
|---|---|---|---|---|
| | | $CRA_2$ | $CRB_2$ | |
| 0 | 0 | 1 | × | 周辺レジスタA |
| 0 | 0 | 0 | × | 方向レジスタA |
| 0 | 1 | × | × | コントロール・レジスタA |
| 1 | 0 | × | 1 | 周辺レジスタB |
| 1 | 0 | × | 0 | 方向レジスタB |
| 1 | 1 | × | × | コントロール・レジスタB |

×＝不定でよい

● テーブル2：コントロール・レジスタのビット機能

| CRA ビット | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| 機能 | $IRQA_1$ | $IRQA_2$ | $CA_2$ コントロール | | | DDRA アクセス | $CA_1$ コントロール | |

| CRB ビット | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| 機能 | $IRQB_1$ | $IRQB_2$ | $CB_2$ コントロール | | | DDRB アクセス | $CB_1$ コントロール | |

DDRA＝方向レジスタA　DDRB＝方向レジスタB

**表3.3[2] パラレル・インターフェース6821の制御信号テーブル**

● テーブル3：$CA_1$, $CB_1$の機能設定，割り込み入力として

| $CRA_1$ ($CRB_1$) | $CRA_0$ ($CRB_0$) | 割り込み入力 $CA_1$($CB_1$) | 割り込みフラグ $CRA_7$($CRB_7$) | 割り込み要求出力 $\overline{IRQA}$, $\overline{IRQB}$ |
|---|---|---|---|---|
| 0 | 0 | ↓でアクティブ | ↓でセット | 出力しない |
| 0 | 1 | ↓でアクティブ | ↓でセット | $CRA_7$ ($CRB_7$) がセットされて"L" |
| 1 | 0 | ↑でアクティブ | ↑でセット | 出力しない |
| 1 | 1 | ↑でアクティブ | ↑でセット | $CRA_7$ ($CRB_7$) がセットされて"L" |

(注) CRAはコントロール・レジスタA，次の数字はビット番号
CRBはコントロール・レジスタB，次の数字はビット番号
↓入力信号の"H"から"L"への立ち下がり
↑入力信号の"L"から"H"への立ち上がり
▶ $CRA_7$は，MPUがデータ・レジスタ(周辺レジスタ)Aをリードするとクリアされる
▶ $CRB_7$は，MPUがデータ・レジスタBをリードするとクリアされる

● テーブル4：$CA_2$, $CB_2$の機能設定，割り込み入力として

| $CRA_5$ ($CRB_5$) | $CRA_4$ ($CRB_4$) | $CRA_3$ ($CRB_3$) | 割り込み入力 $CA_2$($CB_2$) | 割り込みフラグ $CRA_6$($CRB_6$) | 割り込み要求出力 $\overline{IRQA}$($\overline{IRQB}$) |
|---|---|---|---|---|---|
| 0 | 0 | 0 | ↓でアクティブ | ↓でセット | 出力しない |
| 0 | 0 | 1 | ↓でアクティブ | ↓でセット | $CRA_6$ ($CRB_6$) がセットされて"L" |
| 0 | 1 | 0 | ↑でアクティブ | ↑でセット | 出力しない |
| 0 | 1 | 1 | ↑でアクティブ | ↑でセット | $CRA_6$ ($CRB_6$) がセットされて"L" |

(注) ↓入力信号の"H"から"L"への立ち下がり
↑入力信号の"L"から"H"への立ち上がり
▶ $CRA_6$は，MPUがデータ・レジスタ(周辺レジスタ)Aをリードするとクリアされる
▶ $CRB_6$は，MPUがデータ・レジスタBをリードするとクリアされる

### ● テーブル2

コントロール・レジスタが受けもつビットの機能を表で示してあります．

ビット2はレジスタ選択の一部であり，0で方向レジスタ，1では周辺レジスタがセレク

表3.3[2] パラレル・インターフェース 6821 の制御信号テーブル（つづき）

● テーブル5：$CB_2$ の機能設定，出力としてライト・ストローブとして使用されることが多い

| $CRB_5$ | $CRB_4$ | $CRB_3$ | "L"になる条件 | "H"になる条件 |
|---|---|---|---|---|
| 1 | 0 | 0 | MPU がデータ・レジスタBにデータを書き込んだ後の最初のEパルスの立ち上がり | $CB_1$ の入力により $CRB_7$ がセットされたとき |
| 1 | 0 | 1 | 同　上 | MPU がこのポートのアクセスを終了し，次のEパルスの立ち上がり |
| 1 | 1 | 0 | 常に"L" | $CRB_3$ を MPU が1にするまで |
| 1 | 1 | 1 | 常に"H" | $CRB_3$ を MPU が0にするまで |

● テーブル6：$CA_2$ の機能設定，出力としてリード・ストローブとして使用されることが多い

| $CRA_5$ | $CRA_4$ | $CRA_3$ | "L"になる条件 | "H"になる条件 |
|---|---|---|---|---|
| 1 | 0 | 0 | MPU がデータ・レジスタAをリードした後のEパルスの立ち下がり | $CA_1$ の入力により $CRA_7$ がセットされたとき |
| 1 | 0 | 1 | 同　上 | MPU がこのポートのアクセスを終了し，次のEパルスの立ち下がり |
| 1 | 1 | 0 | 常に"L" | MPU が $CRA_3$ を1にするまで |
| 1 | 1 | 1 | 常に"H" | MPU が $CRA_3$ を0にするまで |

トされます．イニシャライズでは，このビットをまず0として方向レジスタに入出力方向を書き込みます．そして次に，再びコントロール・レジスタを選択してほかのビットを必要に応じてセットすると同時に，ビット2も1にセットして，周辺レジスタをアクセス可能にする，という手順が普通です．

### ● テーブル3

$CA_1$ および $CB_1$ の機能が説明されています．この二つの信号は入力のみが可能であり，データ受け渡しのタイミングを取るための信号や，割り込みの入力として使用します．

表で示すように，この入力の機能は，コントロール・レジスタに書き込むビット0とビット1によって決定されます．

### ● テーブル4～6

$CA_2$ および $CB_2$ の機能を設定する方法が述べられています．この二つの信号は，入力また

図3.8[2] シリアル・インターフェースACIA(6850)のピン配列と信号

| ピン | 信号 | 信号 | ピン |
|---|---|---|---|
| 1 | $V_{SS}$ | $\overline{\mathrm{CTS}}$ | 24 |
| 2 | Rx Data | $\overline{\mathrm{DCD}}$ | 23 |
| 3 | Rx Clk | $D_0$ | 22 |
| 4 | Tx Clk | $D_1$ | 21 |
| 5 | $\overline{\mathrm{RTS}}$ | $D_2$ | 20 |
| 6 | Tx Data | $D_3$ | 19 |
| 7 | $\overline{\mathrm{IRQ}}$ | $D_4$ | 18 |
| 8 | $CS_0$ | $D_5$ | 17 |
| 9 | $\overline{CS_2}$ | $D_6$ | 16 |
| 10 | $CS_1$ | $D_7$ | 15 |
| 11 | RS | E | 14 |
| 12 | $V_{DD}$ | R/$\overline{\mathrm{W}}$ | 13 |

Rx Data：シリアル信号の受信入力
Rx Clk ：受信のためのクロック入力．ボーレイトに関係
Tx Clk ：送信のためのクロック入力．ボーレイトに関係．受信と送信のボーレイトが異なることはあまりないので，普通はRx ClkとTx Clkは同じクロックを使用する
$\overline{\mathrm{RTS}}$ ：リクエスト・ツゥ・センド．モデムを送信状態とするための制御信号出力
Tx Data：シリアル信号の送信出力
$\overline{\mathrm{IRQ}}$ ：割り込み要求出力
$CS_0$, $CS_1$, $\overline{CS_2}$：チップ・セレクト
RS ：レジスタ・セレクト
$\overline{\mathrm{CTS}}$ ：クリア・ツゥ・センドの制御入力信号．この信号が"H"のときは，送信レジスタが空を示すフラグ(TDRE)のセットが禁止される
$\overline{\mathrm{DCD}}$ ：データ・キャリア・ディテクトの制御入力信号．モデムからの入力で，"L"ではモデムがキャリア信号を受信していることを示す．受信割り込みが許可されていれば，この信号の立ち上がりで$\overline{\mathrm{IRQ}}$がアクティブになる
$D_0$～$D_7$：データ・バス
E ：Eクロック
R/$\overline{\mathrm{W}}$ ：リード/ライト

は出力として使用でき，コントロール・レジスタのビット3，4および5によって機能を選択します．

入力として使用した場合はどちらも同じ機能をもつことができますが，出力として使用した場合には，一部に機能の差があります．テーブル5および6を注意深く参照してください．

プリンタの出力に使用した例を第5章に示しておきました．プログラム例は，**リスト5.6**および**リスト5.7**を参照してください．

### ◆ 6850

ACIA(Asynchronous Communications Interface)と呼ばれ，RS-232Cインターフェースにはなくてはならないものです．7ビットまたは8ビットのシリアル型式の信号の入出力を行います．

ピン配置と信号名の概要説明を**図3.8**に示しておきます．

内部レジスタの選択は**表3.4**に示すように，RSとR/$\overline{\mathrm{W}}$信号によって行います．RSは通常ではアドレス・バスの$A_0$が入力され，従って，アドレス空間では2バイトしか占めませ

**表3.4**[2]
シリアル・インターフェース 6850
のレジスタ選択

| RS | $R/\overline{W}$ | レジスタ |
|---|---|---|
| 0 | 0 | コントロール・レジスタ |
| 0 | 1 | ステータス・レジスタ |
| 1 | 0 | 送信データ・レジスタ |
| 1 | 1 | 受信データ・レジスタ |

**表3.5**[2] シリアル・インターフェース 6850 のプログラム・レファレンス

● テーブル1：コントロール・レジスタとステータス・レジスタの内容

| ビット | コントロール・レジスタ | ステータス・レジスタ |
|---|---|---|
| 0 | クロックのカウンタ・デバイド，$CR_0$ | 受信データ・レジスタがフル，受信データのリード可 |
| 1 | クロックのカウンタ・デバイド，$CR_1$ | 送信データ・レジスタが空，次のデータ書き込み可 |
| 2 | ワード・セレクト，$CR_2$ | データ・キャリア・ディテクト，$\overline{DCD}$ |
| 3 | ワード・セレクト，$CR_3$ | クリア・ツゥ・センド，$\overline{CTS}$ |
| 4 | ワード・セレクト，$CR_4$ | フレーミング・エラー・フラグ |
| 5 | $\overline{RTS}$ のコントロール | 受信オーバラン・フラグ |
| 6 | $\overline{RTS}$ のコントロール | パリティ・エラー・フラグ |
| 7 | 受信割り込み許可 | 割り込み要求フラグ |

● テーブル2：フォーマットの選択，コントロール・レジスタのビット2，3，4で決定する

| $CR_4$ | $CR_3$ | $CR_2$ | フォーマット |
|---|---|---|---|
| 0 | 0 | 0 | 7ビット＋偶数パリティ＋2ストップ・ビット |
| 0 | 0 | 1 | 7ビット＋奇数パリティ＋2ストップ・ビット |
| 0 | 1 | 0 | 7ビット＋偶数パリティ＋1ストップ・ビット |
| 0 | 1 | 1 | 8ビット＋奇数パリティ＋1ストップ・ビット |
| 1 | 0 | 0 | 8ビット＋2ストップ・ビット |
| 1 | 0 | 1 | 8ビット＋1ストップ・ビット |
| 1 | 1 | 0 | 8ビット＋偶数パリティ＋1ストップ・ビット |
| 1 | 1 | 1 | 8ビット＋奇数パリティ＋1ストップ・ビット |

ん．そのため，同じアドレスでもリードとライトによって，アクセスするレジスタが異なることに注意してください．

**表3.5** では 6850 を使用したプログラムに必要な，レジスタの内容とビットの意味を示しておきます．

**表3.5**[2] シリアル・インターフェース6850のプログラム・レファレンス（つづき）

● テーブル3：送信制御ビットの設定

| $CR_6$ | $CR_5$ | 機能 |
|---|---|---|
| 0 | 0 | $\overline{RTS}$＝"L"レベル，送信割り込み不可 |
| 0 | 1 | $\overline{RTS}$＝"L"レベル，送信割り込み可 |
| 1 | 0 | $\overline{RTS}$＝"H"レベル，送信割り込み不可 |
| 1 | 1 | $\overline{RTS}$＝"L"レベル，送信割り込み不可．送信データをスペース（0レベル）に固定する |

（注）送信割り込み可の状態では，TDREフラグのセット，すなわち送信データ・レジスタが空で$\overline{IRQ}$を発生する

● テーブル4：クロック分周比の設定

| $CR_1$ | $CR_0$ | 機能 |
|---|---|---|
| 0 | 0 | ÷1 |
| 0 | 1 | ÷16 |
| 1 | 0 | ÷64 |
| 1 | 1 | マスタ・リセット |

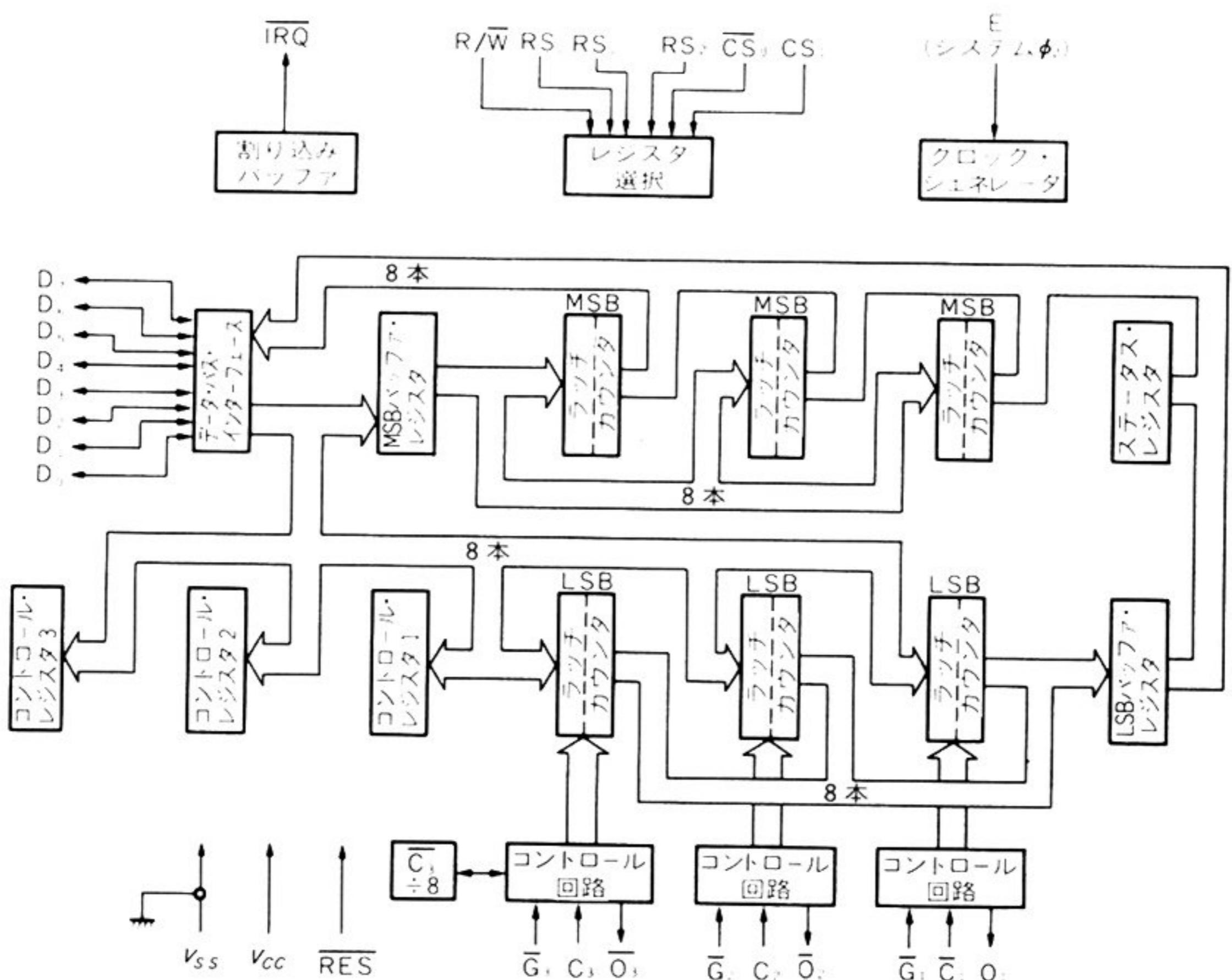
この設定によって分周されたクロック(Tx Clk, Rx Clk)がボーレイトとなる

(例) 1200ボー，÷16とした場合のクロックは，
1200×16＝19200(Hz)

このテーブル4で示されるように，コントロール・レジスタに$03($CR_0$, $CR_1$が1)を書き込んだ場合には，マスタ・リセット機能となり，すべての内部レジスタを初期状態にリセットします．ピン配列で示されるように，このデバイスにはリセットの入力信号がありません．ACIAのイニシャライズ・プログラムでは，テーブル4で示すコードによるリセットを最初に実行しておくのが無難です．

実際のプログラム例は，第5章の**リスト5.1**から**リスト5.5**を参照してください．

### ◈ 6840

PTM(Programmable Timer Module)は，前記の二つ周辺デバイスよりも後から発表されたせいか，内容も機能もはるかに複雑であり，周辺デバイスに不慣れな方は少々覚悟してかかるほうがよいようです．最近ではC-MOS版のHD6340も出回っています．

このタイマ・モジュールは独立した三つの16ビット・カウンタをもち，ワンショット出力や連続した方形波の出力など各種の出力信号の発生，さらにパルス・カウンタ，タイム・インターバルの測定などと多彩な用途に対応できます．

ピン配置とブロック・ダイヤグラムを**図3.9**および**図3.10**に示します．

図3.9[2]
カウンタ/タイマ6840の
ピン配置図

図3.10[2] カウンタ/タイマ6840のブロック・ダイヤグラム

システム・バス信号は，その名称も機能も6821や6850とほとんど同様であるので，説明は省略します．6840特有の入出力信号について説明しておきます．

● **クロック入出力信号($\overline{C_1}$, $\overline{C_2}$, $\overline{C_3}$)**

この三つの入力信号は，タイマ#1,#2,#3のデクリメントを行うための外部クロック入力信号です．

この信号はE信号と非同期でもよいのですが，E信号と同期化されて取り込まれるため，“H”，“L”の期間は少なくても，E信号の1周期と入力セットアップ時間および保持時間を加えた時間が必要です．セットアップ時間と保持時間の和は，1MHzクロックの6840では250nsが必要です．

同期化には3サイクルのEクロックが必要であり，カウンタのデクリメントはEの4クロック目からになりますが，入力信号の周波数には影響なく，入力信号と内部に取り込まれた信号との間にディレイとして残ります．

$\overline{C_3}$はタイマ#3への直接入力だけでなく，内蔵の÷8プリスケーラを通して入力することもできます．

● **ゲート入力信号($\overline{G_1}$, $\overline{G_2}$, $\overline{G_3}$)**

タイマ#1,#2,#3のトリガまたはゲート信号の入力です．

この信号もEクロックによって同期化されるために，“H”，“L”それぞれのレベル幅は，クロック入力信号と同様の時間が必要です．

この信号は内部の16ビット・カウンタに直接作用をするため，$\overline{G_3}$については÷8プリスケーラから独立して動作します．

● **タイマ出力信号($\overline{O_1}$, $\overline{O_2}$, $\overline{O_3}$)**

2個のTTLをドライブ可能な出力信号です．

コンティニュアス・モードまたはワンショット・モードで，プログラムに従った波形を出力します．周波数およびパルス幅比較モードでも波形の出力が行われますが，定義された波形ではありません．

● **レジスタのセレクト**

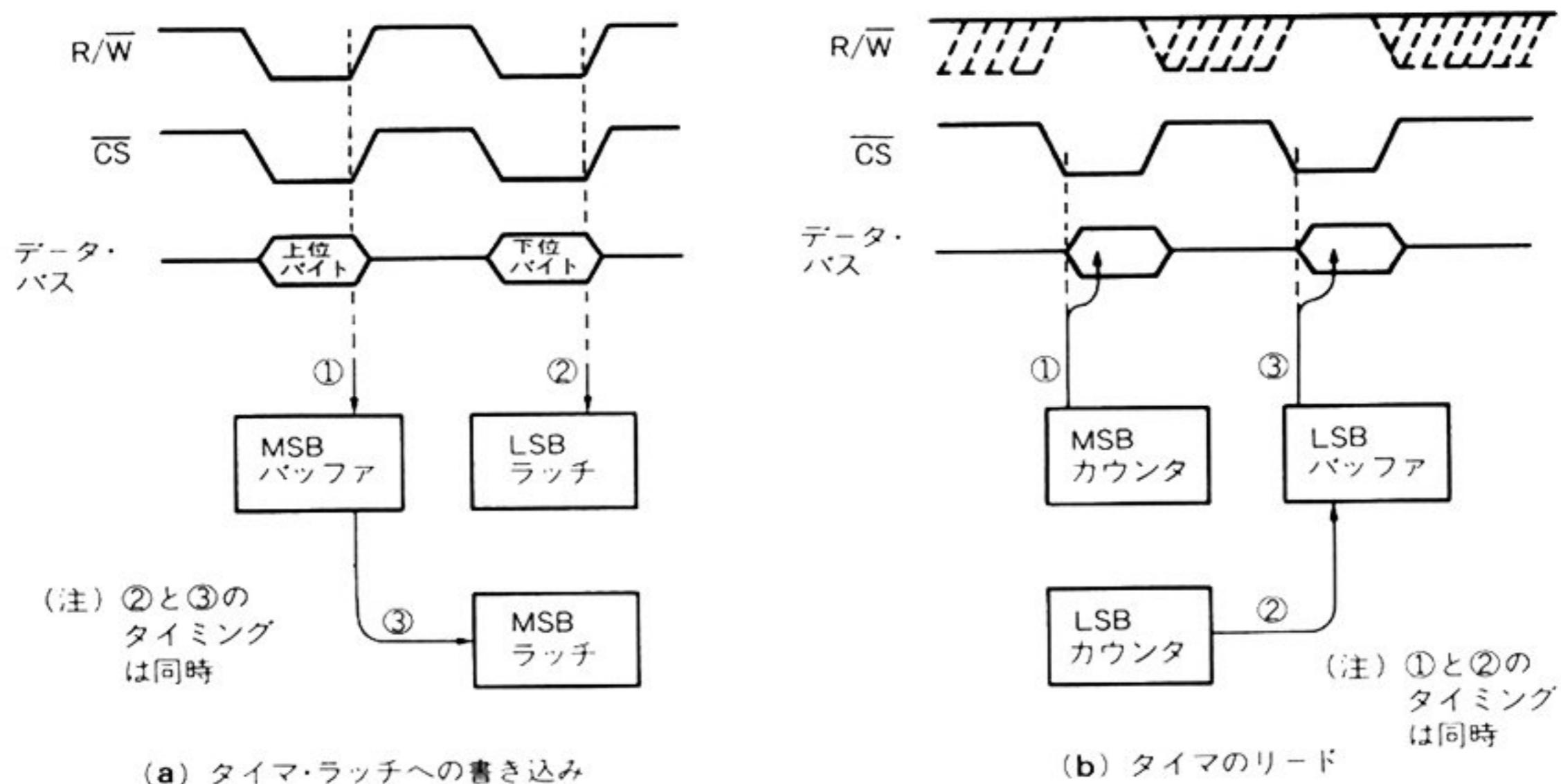
6840の内部レジスタは，アドレス・バスの$A_0$, $A_1$, $A_2$に接続される$RS_0$, $RS_1$, $RS_2$によって選択されます．つまりアドレス空間では8バイトを占めるわけです．

レジスタ・セレクトを表3.6に示しますが，多機能なチップのため，これも少し込み入ったものになっています．

コントロール・レジスタはタイマに対応して三つありますが，アドレスでは二つだけで

表3.6[2] カウンタ/タイマ6840のレジスタ・セレクト

| レジスタ・セレクト | | | 機能 | |
|---|---|---|---|---|
| $RS_2$ | $RS_1$ | $RS_0$ | $R/\overline{W}=0$ | $R/\overline{W}=1$ |
| 0 | 0 | 0 | $CR_{20}=0$のとき，コントロール・レジスタ#3<br>$CR_{20}=1$のとき，コントロール・レジスタ#1 | |
| 0 | 0 | 1 | コントロール・レジスタ#2 | ステータス・レジスタ |
| 0 | 1 | 0 | MSBバッファ | タイマ#1カウンタ |
| 0 | 1 | 1 | タイマ#1ラッチ | LSBバッファ |
| 1 | 0 | 0 | MSBバッファ | タイマ#2カウンタ |
| 1 | 0 | 1 | タイマ#2ラッチ | LSBバッファ |
| 1 | 1 | 0 | MSBバッファ | タイマ#3カウンタ |
| 1 | 1 | 1 | タイマ#3ラッチ | LSBバッファ |

(注) $CR_{20}$：コントロール・レジスタ#2のビット0

す．コントロール・レジスタ#2のビット0は，#1,#3のレジスタを選択します．

16ビット・タイマの読み書きは，バッファ・レジスタとラッチを使って行います．書き込みの場合では，上位のバイトをMSBバッファに書き，次に下位のバイトをタイマ・ラッチに書きます．タイマ・ラッチへの書き込みが終了した時点で，MSBバッファ・レジスタの内容も同時に，対応する16ビット・タイマの上位バイトのラッチへ転送されます．

68系のプロセッサでは16ビット・データを扱う場合に，上位のバイトはアドレスの下位に配置されます．6809の16ビット・データのストア命令は，上位のバイトをまずストアして，次のアドレスに下位のバイトをストアします．ですから，命令の実行と6840のバッファとラッチを使ったタイマの書き込みとは都合よく対応しているのです．

タイマのリードの場合には，下位バイトに対してバッファが使われています．これも命令の実行にうまく対応したものであり，上位バイトのデータである下位のアドレスをリードしたときに，タイマの下位バイトがLSBバッファ・レジスタに転送されます．つまり16ビットのデータを読むとは，8ビットずつの2回に分けて二つのアドレスを読むことになりますが，この時間中に上位桁に影響するデクリメントが発生したとしても，意味のないデータとならないよう考慮されたものです．この様子は，図3.11を参照してください．

レジスタ・セレクトの表を見る限りでは，MSBバッファとLSBバッファは三つずつあるように見えますが，ブロック・ダイヤグラムから見て取れるように，内部ではそれぞれに一つしかありません．ですから三つのバッファにMSBだけを先に書いておき，次に下位

図3.11[2] カウンタ/タイマ6840の16ビット・タイマ読み書きのタイミング

(a) タイマ・ラッチへの書き込み

R/W
CS
データ・バス
①
③
MSB カウンタ
LSB バッファ
LSB カウンタ
②
(注) ①と②のタイミングは同時

(b) タイマのリード

① MSBバッファへ上位バイトをライト
② LSBラッチへ下位バイトをライト
③ ②が発生したときにMSBバッファの内容がMSBラッチに転送される

① MSBカウンタの内容をデータ・バスに乗せる
② ①の発生と同時に，LSBカウンタの内容をLSBバッファに転送する
③ LSBバッファの内容をデータ・バスに乗せる

のデータを三つ書き込むといったプログラムでは具合が悪いことになります．

● コントロール・レジスタとステータス・レジスタ

コントロール・レジスタは表3.7に示すように，ビット1からビット7までは三つのコントロール・レジスタが同じ機能をもっていますが，ビット0はそれぞれによって機能が異なります．

ビット3，4，5は，タイマの多彩な動作モードを決定するビットであり，その詳細は後述の説明と別表(表3.9から表3.12)を参照してください．

ステータス・レジスタは表3.8に示すように，四つの割り込みフラグからなり，残りの4ビットは空きビットです．リードした場合には，この空きビットは0が読み出されます．

ビット0，1，2はタイマ#1,#2,#3の割り込みフラグを表し，ビット7はコンポジット・インタラプト・フラグ，つまり#1,#2,#3の割り込みフラグの論理和が現れます．

表3.7[2] カウンタ/タイマ6840のコントロール・レジスタの機能

● ビット0

| コントロール・レジスタ#1 | | コントロール・レジスタ#2 | | コントロール・レジスタ#3 | |
|---|---|---|---|---|---|
| 0 | すべてのタイマが動作可能 | 0 | コントロール・レジスタ#3を選択 | 0 | タイマ#3のクロックはプリスケーラを使わない |
| 1 | すべてのタイマがプリセット | 1 | コントロール・レジスタ#1を選択 | 1 | タイマ#3のクロックは÷8プリスケーラを使う |

● ビット1～7　#1～#3のタイマ共通

| ビット | | 機能 |
|---|---|---|
| 1 | 0<br>1 | 外部クロック（$\overline{C_x}$）をタイマのクロック・ソースとする.<br>Eクロックをタイマのクロック・ソースとする |
| 2 | 0<br>1 | ノーマル16ビット・カウンティング・モード<br>デュアル8ビット・カウンティング・モード |
| 3，4，5 | | タイマのモード制御．表3.9参照 |
| 6 | 0<br>1 | タイマの割り込みフラグのセットにより$\overline{IRQ}$は出力されない<br>タイマの割り込みフラグのセットにより$\overline{IRQ}$が出力される |
| 7 | 0<br>1 | タイマの出力$\overline{O_x}$はマスクされ出力されない<br>タイマの出力$\overline{O_x}$が出力される |

表3.8[2] カウンタ/タイマ6840のステータス・レジスタ

| ビット | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| 機　能 | INT | / | / | / | / | $I_3$ | $I_2$ | $I_1$ |

$I_1$＝タイマ#1の割り込みフラグ
$I_2$＝タイマ#2の割り込みフラグ
$I_3$＝タイマ#3の割り込みフラグ
INT＝$I_1$, $I_2$, $I_3$の論理和

これらのフラグがセットされている場合，ステータス・レジスタをリードしてからフラグに対応しているタイマをリードすると，このフラグがクリアされます．

### ● タイマの機能

タイマの具体的な機能の説明に入ります．可能な機能は，まず二つに分類され，信号の発生と計測のモードです．

信号の発生では，連続したクロックとワンショット（シングル・ショット）とが可能であり，計測モードでは周波数比較とパルス幅比較とができます．

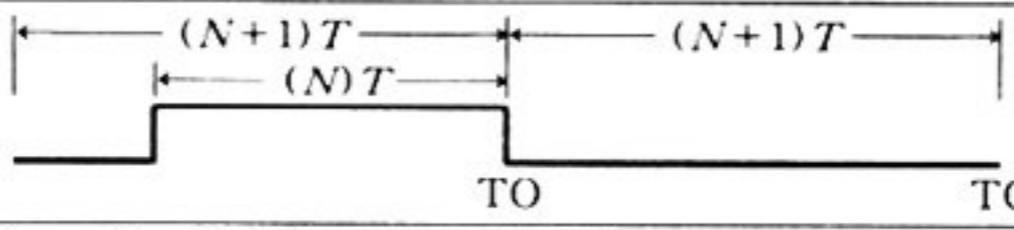
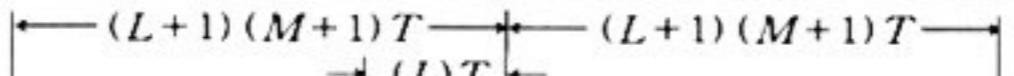
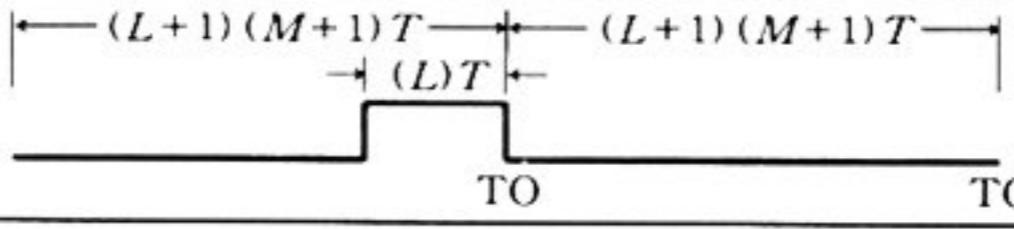
表3.9[2] カウンタ/タイマ6840のタイマの動作モード

| コントロール・レジスタ | | | 動作モード | |
|---|---|---|---|---|
| ビット1 | ビット2 | ビット3 | | |
| 0 | * | 0 | コンティニュアス・モード | シンセサイザ機能 |
| 0 | * | 1 | シングル・ショット・モード | |
| 1 | 0 | * | 周波数比較モード | 計測機能 |
| 1 | 1 | * | パルス幅比較モード | |

*は表3.10，表3.11，表3.12を参照

表3.10[2] カウンタ/タイマ6840のタイマのコンティニュアス・モード

| ビット2 | ビット4 | カウンタ初期化の条件 | タイマ出力 | |
|---|---|---|---|---|
| 0 | 0 | $\overline{G}$↓+W+R | ←(N+1)T→←(N+1)T→←(N+1)T→ TO TO TO | 16ビット・モード |
| 0 | 1 | $\overline{G}$↓+R | | |
| 1 | 0 | $\overline{G}$↓+W+R | ←(L+1)(M+1)T→←(L+1)(M+1)T→ ←(L)T→ TO ←(L)T→ TO | デュアル8ビット・モード |
| 1 | 1 | $\overline{G}$↓+R | | |

G↓＝ゲート入力の立ち下がり
W＝タイマへの書き込み
R＝タイマ・リセット
$N$＝カウンタに書き込んだ値．16ビット
$L$＝下位8ビットの値
$M$＝上位8ビットの値
TO＝タイム・アウト．割り込みフラグがセットされ，カウンタが初期化される
$T$＝クロックの周期

この機能の選択はコントロール・レジスタのビット3，4，5で行い，この関係を表3.9に示します．上記以外のビットも，それぞれの機能についての詳細を決定しますので，コントロール・レジスタのセッティングでは，ほかの表も合わせて参照してください．

### ● コンティニュアス・モード

連続したクロックを発生するモードです．コントロール・レジスタのビット2により，16ビット・モードとデュアル8ビット・モードに分けられます．

16ビット・モードでは，出力が0の時間と1の時間が同じであり，それぞれの時間はラッチに書き込まれた16ビットの値$N$に1を加えてクロックの周期を乗じた長さになります．

表3.11[2] カウンタ/タイマ6840のタイマのシングル・ショット・モード
(ビット3=0, ビット5=1, ビットは各コントロール・レジスタのビット)

| ビット2 | ビット4 | カウンタ初期化の条件 | タイマ出力 | |
|---|---|---|---|---|
| 0 | 0 | $\overline{G}$↓+W+R | $(N+1)T$ / $(N)T$ / $(N+1)T$ | 16ビット・モード |
| 0 | 1 | $\overline{G}$+R | TO TO | |
| 1 | 0 | $\overline{G}$+W+R | $(L+1)(M+1)T$ / $(L)T$ / $(L+1)(M+1)T$ | デュアル8ビット・モード |
| 1 | 1 | $\overline{G}$+R | TO TO | |

デュアル8ビット・モードでは，出力信号のデューティをプログラムで決めることができます．この場合，上位バイト$M$と下位バイト$L$は区別して扱われ，信号の周期は，$M+1$，$L+1$およびクロック周期の積であり，出力が1の時間は$L$とクロック周期の積になります．

コントロール・レジスタのビット4は，カウンタを初期化する条件に関わります．以上の関係を表3.10に示しておきます．

このように設定された信号が出力に現れるためには，対応するコントロール・レジスタのビット7が1になっている必要があります．

### ● シングル・ショット・モード

一つのパルスだけを発生するモードです．このモードにも16ビット・モードとデュアル8ビット・モードとがあります．

コンティニュアス・モードでは，$\overline{G}$が“L”レベルであることが信号出力に必要ですが，シングル・ショット・モードではカウンタのイニシャライズに関わるだけで，出力波形は$\overline{G}$のレベルに依存しません．

ほかの点については，コンティニュアス・モードとほぼ同様です．表3.11を参照してください．

### ● 計測モード

波形の計測では先に述べたように，周波数の比較とパルス幅の比較が行えます．どの場合でも比較判定は，ゲートに入力されたパルス信号と内部のタイマで発生したタイム・インターバルとの間で行われ，その結果は割り込みフラグで示されます．

このモードでは，タイマの出力信号が利用されることはめったにありませんが，次のよ

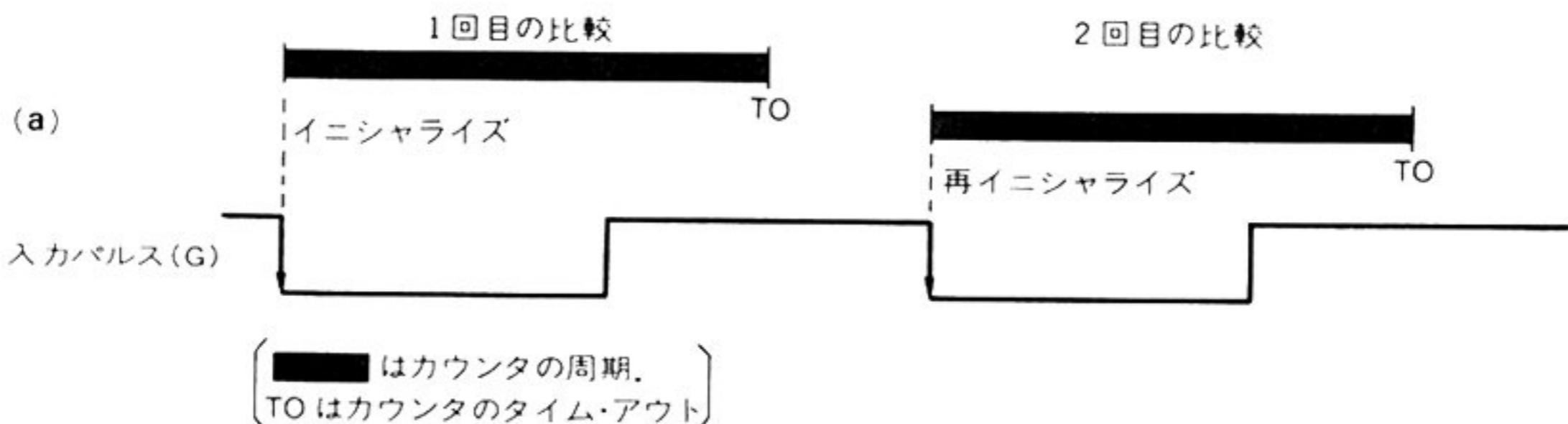
表3.12[2] カウンタ/タイマ6840の計測モード（周波数またはパルス幅の比較）

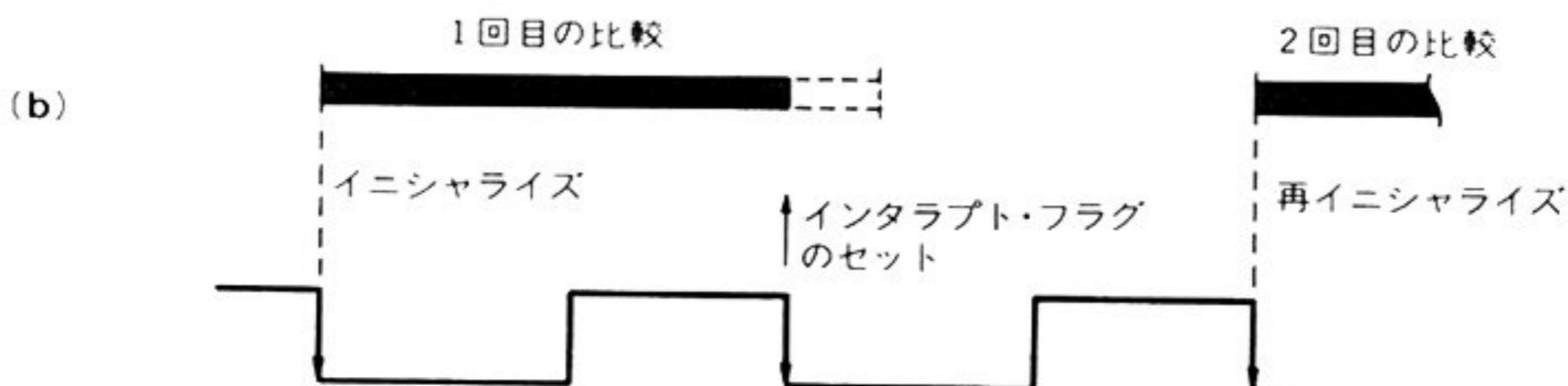
| ビット4 | ビット5 | 機　能 | 割り込みフラグがセットされる条件 |
|---|---|---|---|
| 0 | 0 | 周波数比較 | ゲート入力の周期がカウンタのタイム・アウトより小さい |
| 0 | 1 | 周波数比較 | ゲート入力の周期がカウンタのタイム・アウトより大きい |
| 1 | 0 | パルス幅比較 | ゲート入力の“L”の時間がカウンタのタイム・アウトより小さい |
| 1 | 1 | パルス幅比較 | ゲート入力の“L”の時間がカウンタのタイム・アウトより大きい |

うな条件で出力されます.

タイマのイニシャライズの発生から最初のタイム・アウトが発生するまでの期間は0レベル，タイム・アウトの発生で1レベル，さらに続けてタイム・アウトが発生した場合は，ステートが反転します.

計測モードの場合にも，タイマ・カウンタは16ビット・モードとデュアル8ビット・モードのどちらでも使用できます．このモードの機能とコントロール・レジスタのビットとの関係を表3.12に示します.

周波数比較モードではゲート入力の立ち下がりから次の立ち下がりまでの時間が比較されるのに対して，パルス幅比較モードでは，入力パルスの“L”レベルの時間が比較されます.

周波数比較モードにおける波形の例を図3.12に示しておきます.

第5章では，モータの回転数計測に利用した例を示しておきました．そのプログラムはリスト5.8ですので参照してください.

## 3.3　大容量メモリとRS-232Cインターフェースの対応

紹介したCPUボードの回路を設計した時点では，ROMの主流は4Kバイトの2732でした．8Kバイトの2764は，暫定の仕様書がやっと手に入る頃だったのです.

メモリの大容量化は大変なもので，この原稿を書いている今では，16Kバイトの27128が主流であり，ビット単価も最も安いEP-ROMになっています．この様子ですと，本書が印刷される頃には，27128も時代遅れのチップになっていることと思います.

図3.12[2] 周波数比較モードの波形の例

この例では，入力パルスの周波数が低く，TOが入力パルスの立ち下がりよりも先に発生するので，割り込みフラグはセットされない

この例では，入力パルスの周波数が高く，TOが発生する以前に入力パルスが立ち下がるので，割り込みフラグがセットされる

しかし，あまり心配することはなく，アドレス信号が増えただけであり，信号の機能やタイミングで基本的な違いはありません．

メモリ IC をアドレス・デコードする回路の定石を知っていれば，新しい IC に対応し，変更するのは難しいことではありません．

図3.13 に，ROM は 2764 と 27128, RAM は 6264 に対応したメモリとアドレス・デコーダ部分の回路を示しておきます．

シリアル・インターフェースについては，CPU ボードではカレント・ループになっていますが，RS-232C のほうが一般的ですので，これも回路例として図3.14 に ACIA と RS-232C インターフェースの部分を示しておきます．

RS-232 C では＋12 V と－12 V の電源が必要ですが，+5 V から＋12 V と－12 V を作る回路を一緒に示しておきました．

図3.13　ROM, RAMとアドレス・デコード回路の例（16Kバイト ROM 27128を使用するために，CPUボード回路の一部を変更した例）

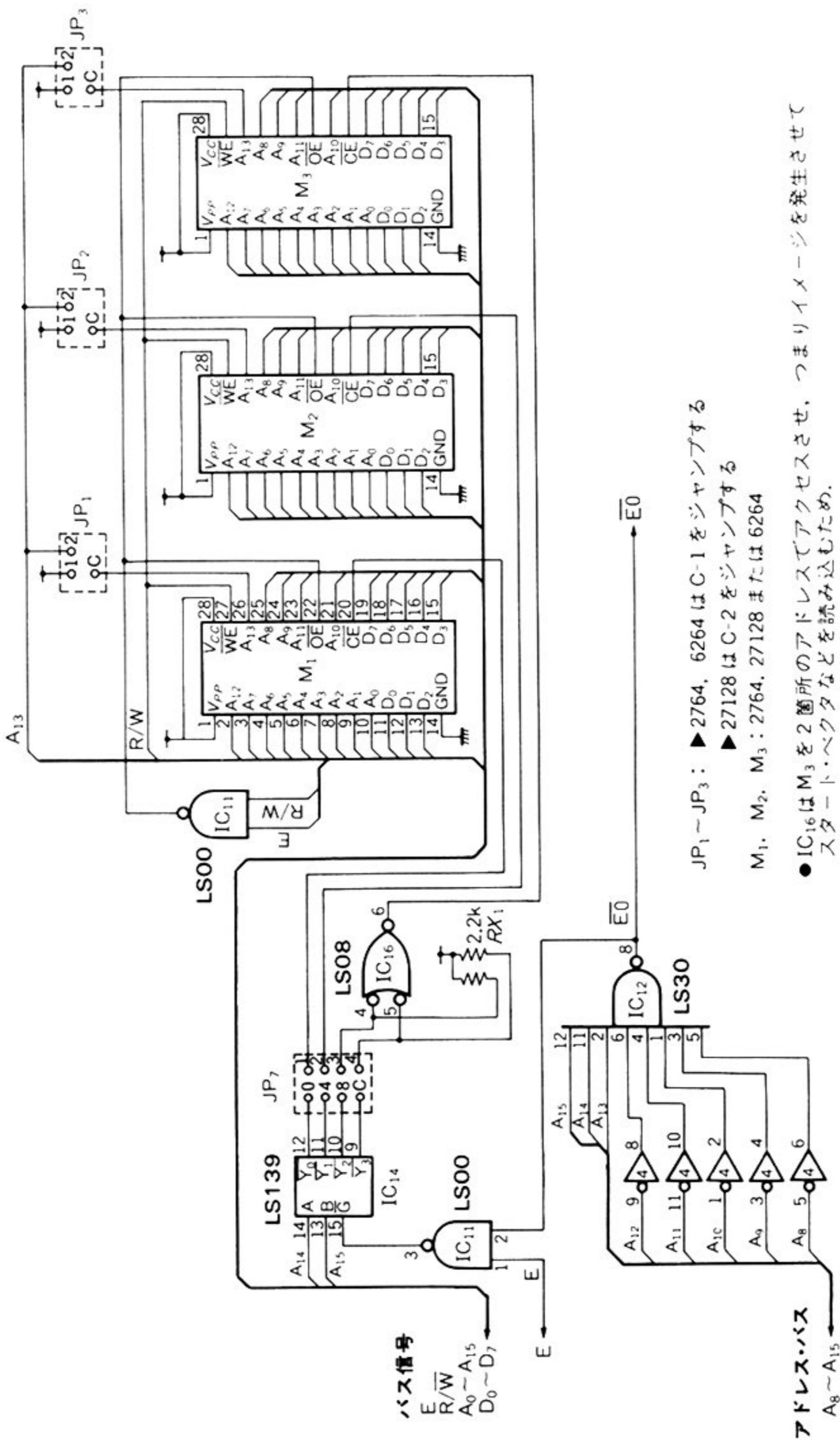
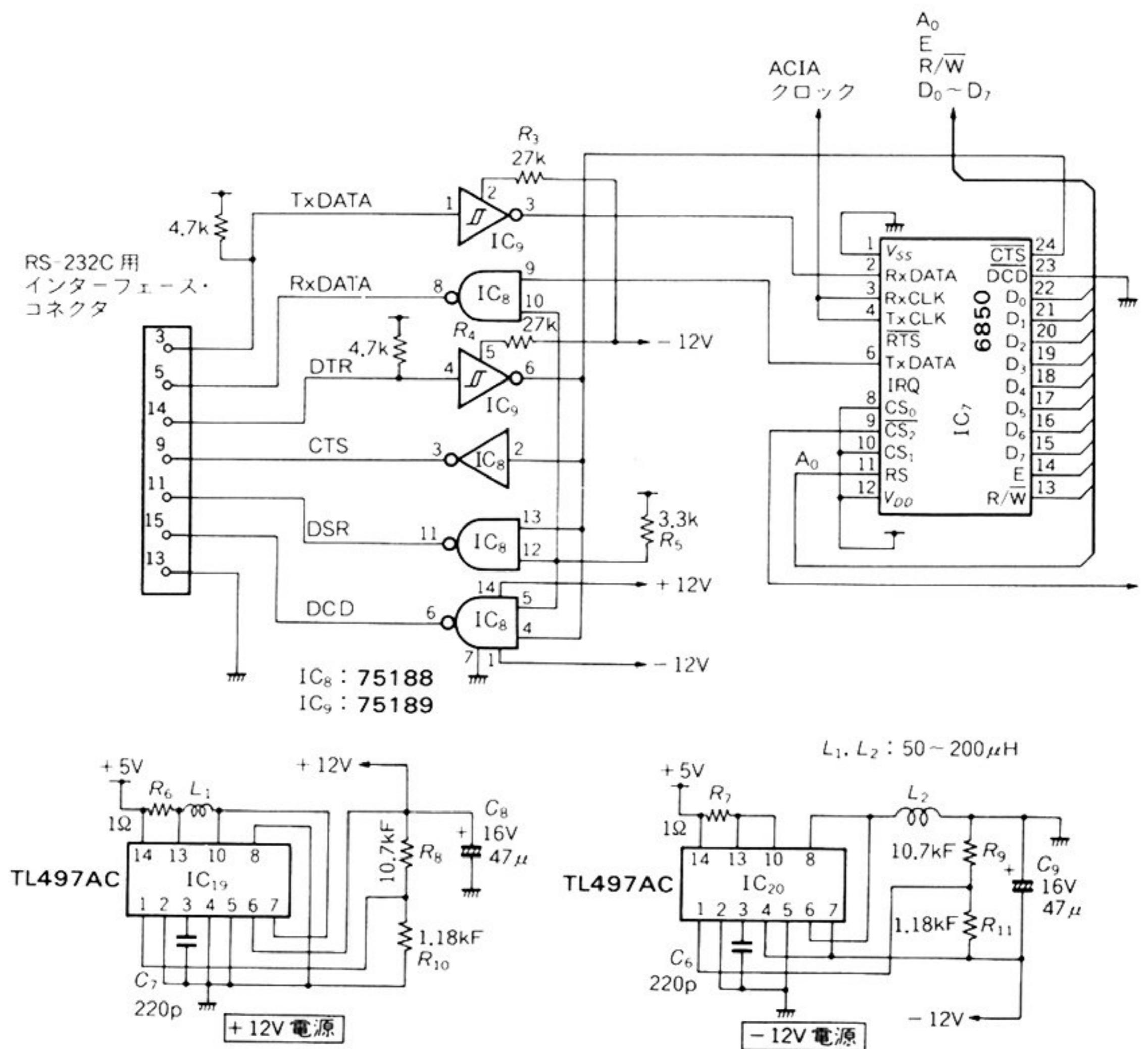



図3.14　RS-232Cインターフェースの例

# 第4章
# 6809のアセンブリ言語と命令

アセンブラによるプログラミングの経験がほとんどない読者のために，アセンブラの基礎知識をまず述べておきます．

アセンブラによるプログラムでは，プロセッサが直接に実行できる命令，つまり機械語とプログラムの記述とが1対1で対応しているのが原則です．

機械語は1か0かの組み合わせであり，これでは私達にとってはなんのことかさっぱりわかりにくいものです．そこで私達の言語に少しでも近いように，機械語に対応させた文字で表現したものがアセンブリ言語であり，アセンブリ言語で書かれたソース・プログラムを機械語に変換する機能をもったものをアセンブラと呼びます．

アセンブリ言語そのものや，この方法の全体もアセンブラと呼ぶことも多いので，ここでは明確な定義による区別をあまりすることなくこの言葉を使用することにします．

アセンブラも言語の一種ですから，それぞれのアセンブラに定められた文法があります．しかしこの文法は，BASICやPASCALなどの文法と比べたら比較にならないくらい単純なものです．

けれども，文法が単純なのとプログラミングが容易なのとはまったく別の問題であり，単純な命令だけで目的を達成するプログラムを作ることは容易ではありません．

ですからアセンブラを学習するとは，プロセッサが実行できる命令を正確に理解することと，単純な命令をいかに組み合わせて目的を達成するかに帰結します．命令をいかに組み合わせるかについては，BASICなどの高級言語でも同様ですが，アセンブラでは最も小さい単位からこれを行わなくてはならないわけです．

## 4.1 アセンブラの文構成

アセンブラ言語で書いた元になるソース文を構成する語で最も重要なものは，ニーモニック，オペランド，ラベル(シンボルとも呼ばれる)の三つに分類されます．ソース・プログラムの例を以下に示します．

```
CHCNV  ADDA  #$30
```

上の例はアセンブラ・プログラムの1行ですが，これが機械語の1命令に対応します．

この例では，ADDAがニーモニックであり，アキュムレータAに加算しろ，ということです．つまりニーモニックとは，私たちの言語に対応させれば動詞ということになります．

後で述べる6809の命令の説明とは，このニーモニックの種類とその機能の説明なのです．

アキュムレータAに何を加算するか，がつぎの #$30であり，これも私たちの言語にすれば目的語に当たり，この部分をオペランドと呼びます．

#はイミディエイト・アドレシングであることを示し，$は16進数を意味しますから，上の例は16進の30をアキュムレータAに加算しろ，ということになります．

もしも#がなかったとすれば，エクステンデド・アドレシングと解釈され，$30はメモリ・アドレスを指し，$30に格納されている内容をアキュムレータAに加算しろ，ということになります．

先頭の語，CHCNVはラベルと呼ばれる部分です．ラベルは機械語に変換されることはありません．プログラム中の特定の箇所を示すために置かれ，ほかの部分ではオペランドとして，ラベルの置かれたアドレス値を示すために使われます．

ラベルはこれ以外にも，疑似命令であるEQUによっても特定の値を定義できます．

オペランドでは簡単な四則演算を記述することもできます．ラベルと演算の組み合わせによるオペランドの例を次に示します．

```
LDX  CHCNV+10
```

この例では，ラベルであるCHCNVの置かれたアドレスに10を加えたアドレスが格納するメモリ内容をインデクス・レジスタXにロードしろ，となります．

当然のことですが，このオペランドでの加算は，アセンブルされたプログラムが実行す

るときに行われるのではなく，アセンブラによってアセンブル時に行われ，定数値としてアセンブルされます．

上記以外の文として，コメントと疑似命令がありますが，その詳細については使用するアセンブラによって一部に異なる点があります．ここでは神戸大学の4氏によって開発され，インターフェース誌1981年2月号で発表していただいた6809セルフ・アセンブラの仕様の範囲で説明します．

● **コメント**

命令行でオペランド(オペランドのない命令ではニーモニック)の後に，スペースで区切られて書かれた文はコメントとみなされます．コメント行とする場合には第1文字を＊にします．

アセンブラは，コメント文とコメント行は読み飛ばします．

● **疑似命令**

疑似命令とは，アセンブラに対しての命令であり，ニーモニックのように機械語に対応したものではありません．

疑似命令の記述法は，ニーモニックの記述と同様であり，ラベルやオペランドを必要とするものもあります．この場合のオペランドは，アセンブル作業の指示だけを意味するものと，オブジェクト・プログラムの一部として，2進数のコードに変換されるものがあります．

以下は疑似命令の説明です．

**END**

ソース・プログラムの終わりを示します．ソース・プログラムが複数のファイルとなる場合でも，それぞれのファイルについてEND文が必要です．

ラベルは付けられません．

**EQU**

ラベルに対して，オペランドの数値を割り当てます．この疑似命令行のオペランドにラベルを使うこともできますが，そのラベルはほかの箇所で明確に数値が定義されていなくてはなりません．

命令の性格からしても明らかですが，ラベルとオペランドは必ず必要です．

**FCB**

1バイトの定数を設定します．コンマで区切って複数バイトの設定もできます．

```
(例)    FCB    $1000−256
        FCB    'E, 04
```

'EはEが文字定数であることを示します.

**FCC**

文字列を定数として設定します.

オペランドは，文字数を指定する定数，コンマ，文字列の順に並べるか，文字列を同じ文字か記号で囲みます.

```
(例)    FCC    3, HOW
        FCC    /TSURUMI/
```

**FDB**

2バイトの定数を設定します．コンマで区切って複数の2バイト単位の定数も設定できます.

**NAM**

オペランドにはプログラムのタイトルを書きます. NAM文はソース・プログラムの第一行目に書きます.

アセンブル・リストには, NAM文で指定されたタイトルがリストの見出しとして表示されます．ラベルは付けられません.

**ORG**

オペランドの定数は，オブジェクト・コードのアドレスを指定します.

この命令に続くステートメントは, ORG命令で指定されたアドレスを先頭アドレスとしてアセンブルされます．必要があれば，プログラム中にいくつあってもかまいません.

ラベルは付けられません.

**PAGE**

リストの改ページを行います．ラベルは付けられません.

**RMB**

オペランドで示すバイト数のメモリ領域を確保します．バイト数を示す数表現は，10進のほかに，$に続く16進数や，@に続く8進数でも可能です.

**SETDP**

アセンブラに対して仮想のダイレクト・ページ・レジスタの値を定義します.

アセンブラは，この値によってダイレクト・アドレシングが可能かどうかを判断しますが，プログラムの実行時においてダイレクト・ページが一致するかどうかはプログラマが判断しておかなくてはなりません．

ラベルは付けられません．

SPC

オペランドの数値だけリストの行送りを行います．ラベルは付けられません．

● **OPT 命令**

この命令もアセンブラを制御する命令です．どのアセンブラでも同一ということではないのですが，神戸大学の4氏によって発表していただいたアセンブラについての説明を以下に引用しておきます．＊印は指定がない場合の初期状態です．

これらの命令は，NAM 文に続いて，プログラムの初めの部分に書きます．

OPT　D ……SETDPの効果あり*
OPT　NOD……SETDPの効果なし
OPT　G ……FCC, FCB, FDBのデータをすべて出力*
OPT　NOG……FCC, FCB, FDBの最初のデータだけ出力
OPT　L ……アセンブル・リストの出力*
OPT　NOL……アセンブル・リストの出力なし
OPT　M ……オブジェクト・コードをメモリへ転送する
OPT　NOM……オブジェクト・コードをメモリへ転送しない*
OPT　O ……オブジェクト・コードを出力する
OPT　NOO……オブジェクト・コードを出力しない*
OPT　P ……改ページ付き*
OPT　NOP……改ページなし
OPT　S ……ラベル・テーブル出力
OPT　NOS……ラベル・テーブル出力なし*

## 4.2 6809の命令

6809の基本命令の数は59しかありません．これは良く整理されたアーキテクチャによるもので，6800の72命令と比べても少ないのです．

しかし，強化されたアドレシング・モードによって最終的な機械語の数，つまりアドレ

シング・モードの違いを含めた命令数は，6800の197に対して6809では1,464に増えています．

このことは，6809は強力なプロセッサであるわりには覚えやすい命令群であり，使いやすいプロセッサであると思います．

以下，基本命令についてアルファベット順に説明しますが，これもすべてを知らなければ，プログラミングができないということではありません．実際のプログラミングにのぞみながら，少しずつボキャブラリを増やしていってください．

命令の実行の結果で影響を受けるコンディション・コード・レジスタの意味についての説明はここでは省略しますが，アセンブラ・プログラミングでは重要なことです．第1章のコンディション・コード・レジスタについての説明(p.12)を参照してください．

### ABX

| ソース型式　ABX |
|---|

アキュムレータBのデータを，符号なし2進数としてインデクス・レジスタXに加算します．

加算命令としては特別な命令であり，16ビットのXと8ビットのBを加算することから，データの演算というよりは，アドレス・ポインタの演算によく使われます．

コンディション・コードは影響を受けません．

### ADC

| ソース型式　ADCA　P；ADCB　P |
|---|

キャリを含む8ビットの加算命令です．オペランドの内容Pにキャリを加え，それをアキュムレータAまたはBに加算します．

コンディション・コードはH, N, Z, V, Cが影響を受けます．

### ADD

| ソース型式　ADDA　P；ADDB　P<br>ADDD　P |
|---|

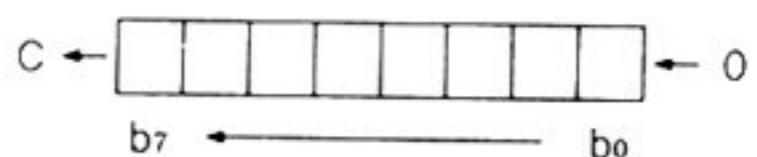
オペランドの内容Pをアキュムレータに加算します．キャリは加えられません．

ADDA, ADDBは8ビットのアキュムレータAまたはBへの加算ですので，オペランドPも8ビットの値ですが，ADDDはAとBを連結したDレジスタへの加算なので，オペランドPもそれにサイズを合わせて16ビット値をとります．

コンディション・コードはH, N, Z, V, Cが影響を受けますが，ADDDではHは影響を受けません．

### AND

| ソース型式 ANDA P；ANDB P |
|---|
| ANDCC P |

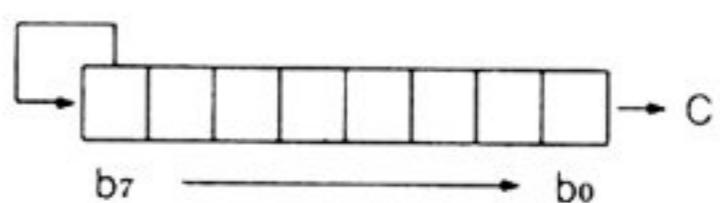
レジスタとオペランドの内容の論理積を取り，結果をレジスタに残します．

ANDAとANDBでは，コンディション・コードのN, Zが演算結果にしたがって影響され，Vはつねにクリアされます．

ANDCCはコンディション・コード・レジスタとの論理積であるので，コンディション・コードは演算結果そのものが残ります．ANDCCのオペランドとしてはイミディエイト・モードだけが可能です．

### ASL

| ソース型式 ASLA ；ASLB |
|---|
| ASL Q |

ASLの動作

アキュムレータAまたはBあるいはオペランドQの全ビットを1ビット分左へシフトします．ビット0は0になり，ビット7はC(キャリ)ビットにシフトされます．

コンディション・コードはN, Z, V, Cが影響を受けます．Hは意味をもちません．

算術左シフトと呼ばれ，論理左シフトLSLとはニーモニックでも区別されていますが，結果的に動作は同じであり，どちらも同じ機械語にアセンブルされます．算術シフトと論理シフトの差は，右シフトの場合に明確に差が出ます．

### ASR

| ソース型式 ASRA ；ASRB |
|---|
| ASR Q |

ASRの動作

アキュムレータAまたはBもしくはオペランドQの全ビットを1ビット分右へシフトします．ビット7の内容は保存され，ビット0はC(キャリ)ビットにシフトされます．

算術右シフトと呼ばれます．論理右シフトLSRとの違いは，ビット7の結果に現れてきます．

2の補数表現をした負のデータに対して論理右シフトを行えば，ビット7(MSB)は0になるため，データは正になってしまいます．このため算術シフトと論理シフトは似ている点がありますが，明確に区別して使用しなければならない場合のあることを理解しなければなりません．

コンディション・コードはN, Z, Cが影響を受けます．Hは意味をもちません．

**Bxx , LBxx**

ブランチ命令については，**表4.1**にまとめておきました．

ブランチ命令は，オペランドの値をオフセット値としてプログラム・カウンタに加算して，プログラムの実行アドレスを変えます．これをブランチするといいます．

表に示すように，ブランチ命令のほとんどは条件つきのブランチであり，これらはコンディション・コードの内容によって，ブランチするかしないかが決まります．

オフセットの語長によって，ショート・ブランチBxxとロング・ブランチLBxxとがあります．ショート・ブランチではオフセットを8ビットの2の補数表現で表し，ロング・ブランチでは16ビットの2の補数表現でオフセット値を表します．

つまり，ロング・ブランチを使用すれば，64Kのメモリ空間のどこへでもブランチできます．

**表4.1**では大まかな使われ方によって4種類に分けてあり，一部にだぶって示されている命令もあります．表で示す条件とは，ブランチする条件となるコンディション・コードの状態を示しています．

シンプルな条件付きブランチに分類されている命令は，コンディション・コードのどれか一つのビットをブランチの条件としているものです．

符号付き条件判断によるブランチは，2の補数表現による符号付き2進数の比較判断に用いるべき命令です．そのため条件の欄に見るように，コンディション・コードのビットの組み合わせは最も複雑です．

符号なし条件判断によるブランチは，符号なし2進数の比較判断によるブランチに用います．

表4.1[2] 6809のブランチ命令

| ニーモニック | 動　作 | 条　件 |
|---|---|---|
| シンプルな条件付きブランチ | | |
| BEQ, LBEQ | 等しいならばブランチ | Z = 1 |
| BNE, LBNE | 等しくなければブランチ | Z = 0 |
| BMI, LBMI | 符号が負ならばブランチ | N = 1 |
| BPL, LBPL | 符号が正ならばブランチ | N = 0 |
| BCC, LBCC | キャリがクリアされていればブランチ | C = 0 |
| BCS, LBCS | キャリがセットされていればブランチ | C = 1 |
| BVS, LBVS | オーバフローが発生していればブランチ | V = 1 |
| BVC, LBVC | オーバフローが発生していなければブランチ | V = 0 |
| 符号付き条件判断によるブランチ | | |
| BGT, LBGT | 大ならばブランチ | $Z \vee (N \oplus V) = 0$ |
| BGE, LBGE | 大もしくは等しいならばブランチ | $N \oplus V = 0$ |
| BEQ, LBEQ | 等しいならばブランチ | Z = 1 |
| BLE, LBLE | 小もしくは等しいならばブランチ | $Z \vee (N \oplus V) = 1$ |
| BLT, LBLT | 小ならばブランチ | $N \oplus V = 1$ |
| 符号なし条件判断によるブランチ | | |
| BHI, LBHI | 大ならばブランチ | $C \vee Z = 0$ |
| BHS, LBHS | 大もしくは等しいならばブランチ | C = 0 |
| BEQ, LBEQ | 等しいならばブランチ | Z = 1 |
| BLS, LBLS | 小もしくは等しいならばブランチ | $C \vee Z = 1$ |
| BLO, LBLO | 小ならばブランチ | C = 1 |
| その他のブランチ | | |
| BSR, LBSR | サブルーチンへブランチ．<br>もどり番地をSスタックでプッシュしてブランチ | |
| BRA, LBRA | 無条件でブランチ | |
| BRN, LBRN | ブランチしない | |

記号の説明：$\vee$＝論理和（OR）
　　　　　$\oplus$＝排他的論理和（Exclusive OR）

## BIT

| ソース型式 BITA P；BITB P |
|---|

アキュムレータのビット・テストを行います．アキュムレータAまたはBの内容とオペランドPとの論理積を取り，その結果がコンディション・コードに反映します．

アキュムレータの内容は変化しません．

ビット・テストという概念ですが，オペランドはアキュムレータのビット番号を示すのではなく，ビット・パターンを示します．ですからこの概念からすれば，1もしくは複数ビットのテストということになります．

コンディション・コードはN，Zが結果による影響を受け，Vはつねにクリアされます．

## CLR

| ソース型式 | CLRA ；CLRB |
|---|---|
| | CLR Q |

アキュムレータAまたはBあるいはオペランドQで示すメモリ内容の全ビットを0にします．コンディション・コードはZがつねにセットされ，N，V，Cはつねにクリアされます．

## CMP

| ソース型式 | CMPA P；CMPB P |
|---|---|
| | CMPD P；CMPS P |
| | CMPU P；CMPX P |
| | CMPY P |

レジスタの内容とオペランドPで示す内容とを比較して，その結果はコンディション・コードに反映されます．ソース型式で示すように，プログラム・カウンタとコンディション・コード・レジスタを除くすべてのレジスタで行えます．

オペランドの語長は，レジスタのサイズに合わせて8ビットまたは16ビット長になります．

この場合の比較とは，レジスタの内容をオペランドで示す内容で減算することに等価で

すが，レジスタとメモリの内容はそのまま保存され，変化しません．

コンディション・コードはN，Z，V，Cが影響を受けます．CMPA，CMPBではHも変化することがありますが意味をもちません．

COM

| ソース型式 | COMA ；COMB |
|---|---|
| | COM Q |

アキュムレータAまたはBあるいはオペランドQで示されるメモリ内容の全ビットを反転します．すなわち論理否定(1の補数)に置き換えます．

コンディション・コードはN，Zが結果による影響を受け，Vはつねにクリア，Cはつねにセットされます．

CWAI

| ソース型式 | CWAI #$xx |
|---|---|

コンディション・コード・レジスタとオペランドであるイミディエイト・データとの論理積を取り，すべての内部レジスタをSスタックで退避してから割り込み待ちの状態になります．

プログラムを割り込み待ちとする場合にこの命令を実行しておけば，割り込みが発生したときにレジスタを退避させる必要がないので，それだけ割り込みサービス・ルーチンを早く起動させることができます．

コンディション・コードと論理積を取るのは，インタラプト・マスク・ビットのIとFをクリアするためです．コンディション・コードはEを除いて論理積の結果で定まり，Eはつねにセットされます．

Eのセットは論理積と同時に行われ，すなわちレジスタの退避よりも先に行われます．このことは，すべてのレジスタを退避したことを示し，割り込みサービス・ルーチンのRTI命令では回復したコンディション・コードのEビットによってすべての内部レジスタが自動的に回復します．

つまりこの場合の割り込み処理では，$\overline{\text{FIRQ}}$であってもすべての内部レジスタが退避そして回復することを意味します．

## DAA

| ソース型式　DAA |
|---|

BCDデータの加算ではこの命令が必要です．

BCDの1桁(4ビット)をニブルと呼びますが，ニブルは0から9の範囲でなくてはなりません．

バイト単位の2進加算を実行した後ではこの条件を満たさないので，正しいBCDデータとするためには，補正を加える必要があるわけです．

この命令はアキュムレータAのデータについてニブルの値とHフラグのテストの結果を使用して10進補正が行われます．

コンディション・コードはN，Z，Cが結果による影響を受け，Vは不定です．

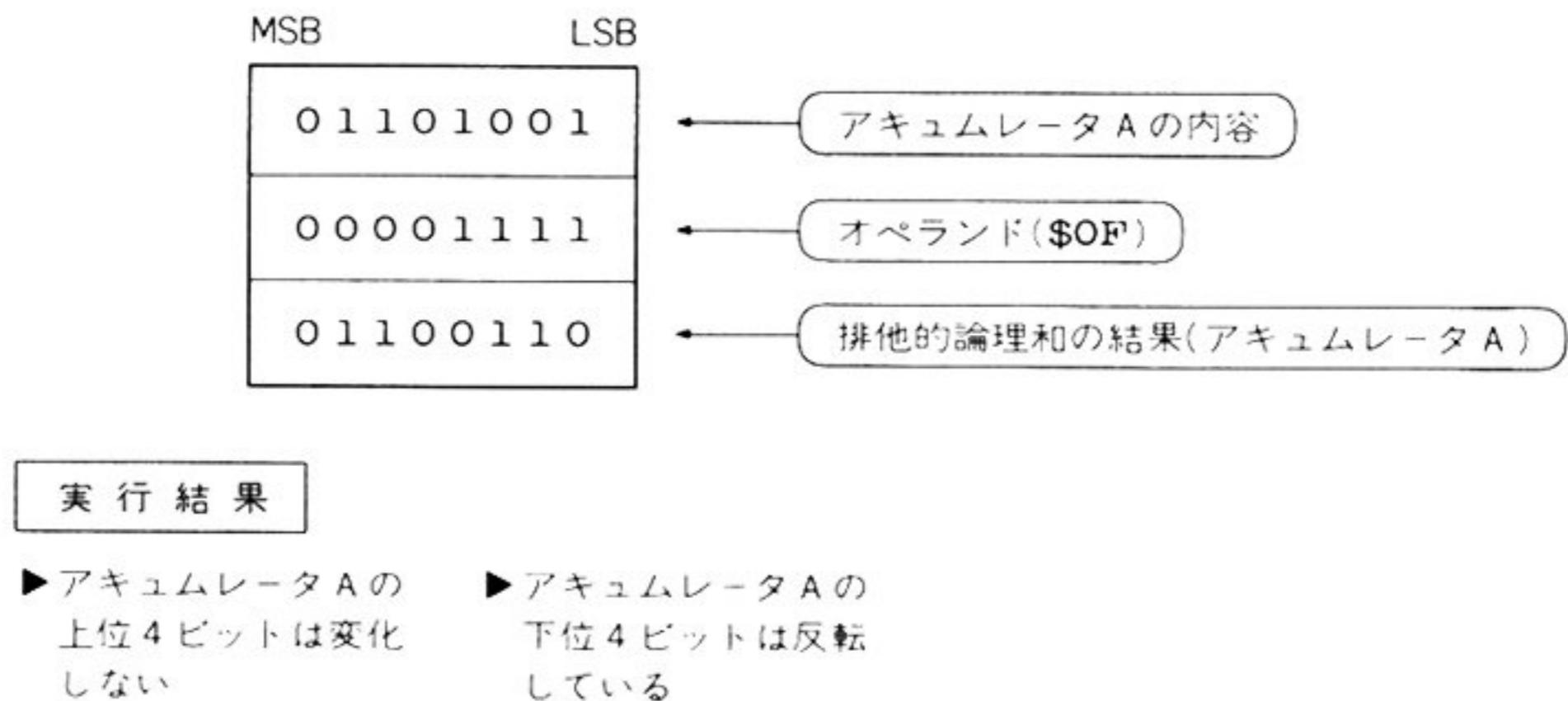
## DEC

| ソース型式　DECA　；DECB<br>　　　　　　DEC　Q |
|---|

アキュムレータAまたはBあるいはオペランドQで示すメモリの内容から1を減じます．

コンディション・コードはN，Zが結果による影響を受け，Vはつねにクリアされます．

この命令ではキャリ・ビットCは影響を受けることがなく，多精度計算のループ・カウンタとして使用できるように考慮されたものです．そのため，減算命令のSUBを使用して1を減じる場合とでは，コンディション・コードに与える影響が異なるので注意してください．

## EOR

| ソース型式　EORA　P；EORB　P |
|---|

アキュムレータAまたはBとオペランドPで示す内容との排他的論理和を取り，結果がアキュムレータに残ります．

コンディション・コードはN，Zが結果による影響を受け，Vはつねにクリアされます．

データの一部のビットのみを反転させるときに，この命令が使えることを知っておくと便利なことがあります．EORA　#$0Fは，アキュムレータAの下位4ビットのみを反転させます．排他的ORの真理値表を作ってみれば明らかです(図4.1参照)．

図4.1　EORA　#$0Fの実行の様子

実行結果

▶アキュムレータAの上位4ビットは変化しない

▶アキュムレータAの下位4ビットは反転している

## EORを使用して一部のビットのみを反転

排他的論理和(EOR)では，1に対しては，レベルが反転，0に対しては変化なしという結果になります．この性格を利用すれば，一部のビットのみを反転させることができるわけです．

すなわち，反転させたいビットの位置だけを1にしたデータと目的のデータとの排他的論理和を取れば，希望するビットのみが反転します．

EORA #$0Fの実行例を図4.1に示します．この例は，アキュムレータAの下位4ビットのみを反転させようという場合です．

アキュムレータAのビット・パターンをどのように入れ替えても下位4ビットのみが反転されることは，この図から容易に理解できると思います．

排他的論理和による一部のビットのみの反転は，高級言語を使用してビット処理のプログラムを作る場合にも役に立つことが多いので，覚えておくとよいでしょう．

レベルは問題にせず，信号が反転したことだけを検出する場合にも，EORがたいへん便利に使用できます．つまり，取り込んだデータと一つ前のデータとのEORを行い，1の立ったビットがあれば，それが反転したビットです．

## EXG

| ソース型式 | EXG R1, R2 |
|---|---|

R1, R2で指定する二つの内部レジスタの内容を交換します．すべてのレジスタについて可能ですが，同一ビット長のレジスタに限られます．

コンディション・コードは影響を受けませんが，コンディション・コード・レジスタについて交換が行われる場合はその限りでありません．

## INC

| ソース型式 | INCA ;INCB |
|---|---|
| | INC Q |

アキュムレータAまたはBあるいはオペランドQで示すメモリの内容に1を加えます．

キャリ・ビットCは影響を受けないので，DECと同様に多精度計算のループ・カウンタとして使用できます．しかし，加算命令のADDまたはADCを使用して1を加えた場合とでは，コンディション・コードに与える影響が異なります．

コンディション・コードはN, Z, Vが影響を受けます．

## JMP

| ソース型式 | JMP EA |
|---|---|

オペランドで示される実効アドレスEAにジャンプします．

この命令で行われる内容は，プログラム・カウンタの内容を実効アドレスの値に書き換えるということです．

コンディション・コードは影響を受けません．

## JSR

| ソース型式 | JSR EA |
|---|---|

プログラム・カウンタPCの内容をもどり番地としてSスタックで退避して，実効アドレスEAにジャンプします．

サブルーチンへのジャンプとして使用されます．サブルーチンの最後では，RTS命令の実行でスタックからプログラム・カウンタがもどされ，JSRが置かれた次のアドレスに実行が移ります．

コンディション・コードは影響を受けません．

LD

| ソース型式 | LDA P；LDB P |
|---|---|
| | LDD P；LDX P |
| | LDY P；LDS P |
| | LDU P |

オペランドPで示す内容を，指定されたレジスタにロードします．ソース型式で示すように，プログラム・カウンタとコンディション・コード・レジスタを除くすべてのレジスタが対象になります．

オペランドの内容は，レジスタの語長に合わせて8または16ビットを取ります．

コンディション・コードはN，Zが結果による影響を受け，Vはつねにクリアされます．

LEA

| ソース型式 | LEAX ；LEAY |
|---|---|
| | LEAS ；LEAU |

上記のソース型式ではオペランドが省略されていますが，インデクスト・アドレシング・モードだけです．

ポインタ・レジスタのX，Y，U，Sが対象であり，オペランドの実効アドレスを計算し，指定されたレジスタに格納します．16ビット・アドレス値の計算をポインタ・レジスタの上で直接に行えるということであり，この機能も6809の特徴の一つです．豊富なアドレシング・モードと相まって強力なデータ処理を実現します．

コンディション・コードへの影響は，LEAXとLEAYの場合のみZが影響を受け，それ以外の影響は与えません．

注意として，LEAX X＋とした場合には，Xレジスタが1加算されるのでなく変化しません．これはプロセッサ内部での実効アドレスの計算順序によるもので，Xを1加算す

るのであれば，

LEAX 1, X

を使用します．

この命令は独特な命令でもあり，不慣れな方にはわかりにくい点があるかも知れません．以下に例を示しておきます．この例の内容から理解してください．

(1) LEAX 10, X

(2) LEAY $300, Y

(3) LEAU 2, X

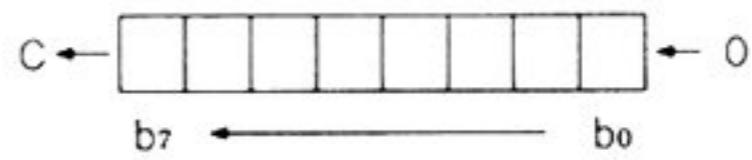
(4) LEAX B, X

(5) LEAS −10, S

(6) LEAS 10, S

(1) の例は，Xレジスタは10を加算した値になります．

(2) の例は，Yレジスタは16進数300を加算した値になります．

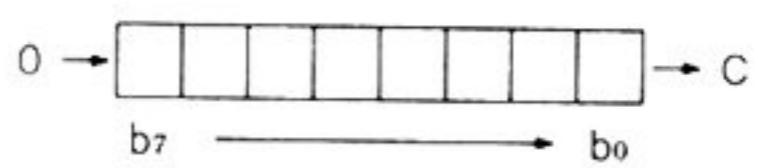
(3) の例は，Xレジスタに2を加算した値がUレジスタに入ります．変化するのはXではなくUレジスタです．

(4) の例は，XレジスタにアキュムレータBが加算されます．Bは符号付き2進数として扱います．ABXと似ていますが，ABXではBの内容を符号なし2進数とします．

(5) の例は，Sスタックを10で減算します．スタック領域に一時的なローカル変数を割り付けるのによく利用され，使用が終れば (6) の例のようにスタックを減算した数だけ加算して元にもどしておきます．

## LSL

ソース型式 LSLA ;LSLB
LSL Q

LSLの動作

アキュムレータAまたはBあるいはオペランドQで示すメモリ内容の全ビットを1ビット分左へシフトします．ビット0は0になり，ビット7はC(キャリ)ビットにシフトされます．

論理左シフトと呼ばれ，算術左シフトのASLとは区別されていますが，機械語のレベルでは同じになります．

コンディション・コードはN, Z, V, Cが影響を受けます.

**LSR**

| ソース型式　LSRA　；LSRB |
|---|
| 　　　　　　LSR　Q |

アキュムレータAまたはBあるいはオペランドQで示すメモリ内容の全ビットを1ビット分右へシフトします. ビット0はC(キャリ)ビットにシフトされ, ビット7へは0が入ります.

論理右シフトと呼ばれ, 算術右シフトとはビット7の扱いが異なります. ASRの説明を参照してください.

コンディション・コードはZ, Cが結果による影響を受け, Nはつねにクリアされます.

**MUL**

| ソース型式　MUL |
|---|

アキュムレータAとBの内容を符号なし2進数とみなして乗算を行い, 結果をアキュムレータDに格納します. アキュムレータDとはAとBを連結して16ビット・アキュムレータとしたもので, Aを上位バイト, Bを下位バイトとして使います.

コンディション・コードはZ, Cが影響を受けますが, この場合のCは特別であり, アキュムレータBのビット7が1ならばセットされ, 0ならばクリアされます.

このCの内容は, 上位バイトを丸めて近似値として取り出す場合に利用され, 次の例のようにプログラムすれば, 上位バイトの近似値がアキュムレータAに残ります.

```
(例)    MUL
        ADCA    #0
```

**NEG**

| ソース型式　NEGA　；NEGB |
|---|
| 　　　　　　NEG　Q |

アキュムレータAまたはBあるいはオペランドQで示すメモリの内容を2の補数値に変

換します．変換前の値を $M$ とすれば，$0-M$ の結果に書き換えます．2進数の実際の処理としては，$\overline{M}+1$ つまり全ビットを反転して1を加えます．

コンディション・コードはN，Z，V，Cが影響を受けますが，V，Cについては，この命令では以下のように特有の意味をもちます．

V　オペランドの内容が$80(−128)のときにセットされます．オペランドの内容は実行後も変化しません．

C　ボローが発生するとセットされます．この命令は0からの減算と考えるので，キャリの反転値を格納してボローを意味させます．

Cがクリアされるのはオペランドの内容が00のときだけです．この場合もオペランドの内容は実行後も変化しません．

### NOP

| ソース型式　NOP |
|---|

プログラム・カウンタをインクリメントするだけで何もしません．コンディション・コードにもなんの影響も与えません．

アセンブラやデバッグ・ツールに不自由していた時代では，プログラムのあちこちにNOPをばらまいておいて，パッチに備えたものですが，便利なツールが気楽に使用できるようになった現在ではその意味も薄れてきました．

### OR

| ソース型式　ORA　P；ORB　P<br>ORCC　#xx |
|---|

ORA，ORBではアキュムレータAまたはBとオペランドPの内容との論理和を取り，結果がアキュムレータに残ります．

コンディション・コードはN，Zが結果による影響を受け，Vはつねにクリアされます．

ORCCはコンディション・コード・レジスタについてのOR操作であり，オペランドはイミディエイト・モードに限られます．

## PSHS

ソース型式　PSHS　レジスタ・リスト

レジスタ・リストにしたがって，一つまたは複数のレジスタをＳスタックで退避します．自身であるＳスタックは退避できません．

レジスタ・リストは，コンマで区切ってレジスタ名を並べます．順序は自由ですが，命令が実行されたときに退避されるレジスタの順は決まっており，以下のとおりです．

PC，U，Y，X，DP，B，A，CC

退避の順 ──→

ソース型式　PSHS　X，Y，D

コンディション・コードは影響を受けません．

一つのレジスタだけを退避するのであれば，オート・デクリメント・モードによるストア命令が使えます．ただしこの場合には，コンディション・コードに影響を与えます．

下に示す２行の例では，アキュムレータＡの内容がＳスタックで退避されることでは同じ結果になりますが，(2)の例では，アキュムレータＡの内容に従ってコンディション・コードが変化します．

```
(例)    (1)  PSHS   A
        (2)  STA    ,-S
```

## PSHU

ソース型式　PSHU　レジスタ・リスト

レジスタ・リストにしたがって，一つまたは複数のレジスタをＵスタックで退避します．自身であるＵスタックは退避できません．

レジスタ・リストはコンマで区切ってレジスタ名を並べます．順序は自由ですが命令が実行されたときに退避されるレジスタの順は決まっており以下のとおりです．

PC，S，Y，X，DP，B，A，CC

退避の順 ──→

コンディション・コードは影響を受けません．

一つのレジスタだけを退避するのであれば，オート・デクリメント・モードによるストア命令が使えることはPSHSの場合と同様ですが，やはりコンディション・コードについては異なります．

```
(例)  (1)  PSHU   A, X, CC
      (2)  PSHU   Y
      (3)  STY    ,--U
```

(1)は複数レジスタの例，(2)と(3)の例は，コンディション・コードへの影響以外は同じ内容です．

## PULS

| ソース型式　PULS　レジスタ・リスト |
|---|

レジスタ・リストにしたがって，一つまたは複数のレジスタをSスタックから回復します．自身であるSレジスタは回復できません．

レジスタ・リストはコンマで区切ってレジスタ名を並べます．順序は自由ですが，命令が実行されたときに回復されるレジスタの順は決まっており以下のとおりです．

PC，U，Y，X，DP，B，A，CC

←――― 回復される順

PSHSの退避の順とちょうど逆になっていることに注意してください．このためレジスタ・リストの内容が同じであるかぎり，PSHSで退避したレジスタはPULSによってそっくり元通りに回復します．

コンディション・コードは影響を受けませんが，CCが回復された場合には回復した内容になります．

一つのレジスタだけを回復するのであれば，オート・インクリメント・モードによるロード命令が使えますが，その場合にはコンディション・コードに影響を与えます．下に示す例では，Dを回復することについては同じですが，(2)の例では，アキュムレータDの内容にしたがってコンディション・コードが変化します．

```
(例)  (1)  PULS   D
      (2)  LDD    ,S++
```

レジスタ・リストにはプログラム・カウンタPCも含むことができること，PCは最後にスタックから取り出されることに注目すれば，サブルーチンのリターンにおいてレジスタの回復とRTS命令を1命令ですますことができます.

下の(1)と(2)の例は，同じ結果になります. 6809のサブルーチンでは，必ずしもRTS命令で終わるとは限らないことに注意しておいてください.

```
(例)  (1)  PULS   Y
           RTS
      (2)  PULS   Y, PC
```

**PULU**

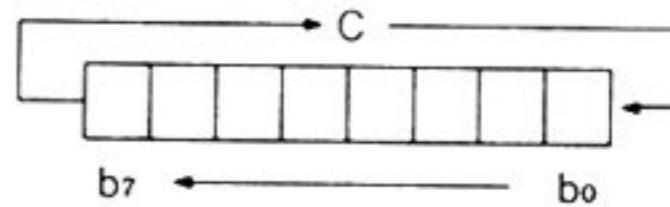
ソース型式　PULU　レジスタ・リスト

レジスタ・リストにしたがって，一つまたは複数のレジスタをUスタックから回復します. 自身であるUレジスタは回復できません.

レジスタ・リストは，コンマで区切ってレジスタ名を並べます. 順序は自由ですが，命令が実行されたときに回復されるレジスタの順は決まっており，以下のとおりです.

PC，S，Y，X，DP，B，A，CC

←――― 回復される順

コンディション・コードは影響を受けませんが，CCが回復された場合は回復された内容になります.

一つのレジスタだけの回復についてもPULSの場合と同様です. 例を下に示しておきます.

```
(例)  (1)  PULU   B
      (2)  LDB    , U+
```

**ROL**

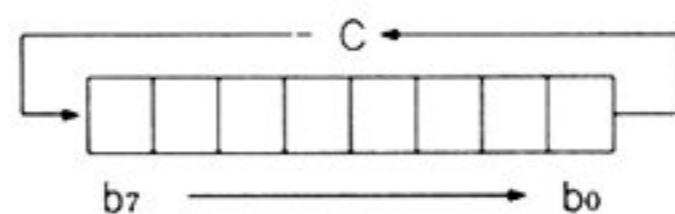
ソース型式　ROLA；ROLB
　　　　　　ROL　Q

アキュムレータAまたはBあるいはオペランドQで示すメモリ内容のすべてのビットを，C(キャリ)を通して1ビット左へローテイトします.

コンディション・コードはN, Z, V, Cが影響を受けます．VとCは次のとおりです．

V　命令実行前のビット6とビット7の排他的論理和が格納されます．

C　命令実行前のビット7の値が格納されます．

語長の長い左シフトを行う場合には，シフト命令とこの命令の組み合わせで行えます．下の例はアキュムレータDの1ビット左シフトの場合です．

```
(例)    LSLB
        ROLA
```

**ROR**

| ソース型式 | RORA　;RORB<br>ROR　Q |
|---|---|

ROR の動作

アキュムレータAまたはBあるいはオペランドQで示すメモリ内容のすべてのビットを，C(キャリ)を通して1ビット右へローテイトします．

コンディション・コードはN, Z, Cが影響を受けます．Cには命令実行前のビット0の値が格納されます．

語長の長い右シフトを行う場合には，シフト命令とこの命令の組み合わせで行えます．下の例はXレジスタで示す2バイトについて，1ビットの右論理シフトの例です．

```
(例)    LSR     ,X
        ROR     1,X
```

**RTI**

| ソース型式 | RTI |
|---|---|

割り込みサービス・ルーチンからの復帰を行う命令です．

回復したCCRのE(エンタイア)ビットをテストして，クリアされていればコンディション・コードとリターン・アドレスがSスタックから回復され，セットされていればSレジスタを除くすべてのレジスタがSスタックから回復されます．

これは，$\overline{\text{FIRQ}}$による割り込みとほかの割り込みとでは，割り込みの発生時に退避するレジスタが異なるためだからです．$\overline{\text{FIRQ}}$ではエンタイア・ビットをクリアして，コンディション・コード・レジスタとプログラム・カウンタだけが退避されますが，$\overline{\text{NMI}}$，$\overline{\text{IRQ}}$では

エンタイア・ビットをセットして，Sレジスタを除くすべての内部レジスタを退避させます．

すなわち，回復すべきレジスタはRTIによって判断されるので，どの割り込みのサービス・ルーチンからの復帰も同じRTIが使用できます．

Sスタックからは，下に示す順でレジスタが回復されます．テストするエンタイア・ビットとは，下に示すように最初に回復するCCに含まれているものです．

エイタイア・ビットがクリアされている場合，

CC，PC

———→ 回復の順

エンタイア・ビットがセットされている場合，

CC，A，B，DP，X，Y，U，PC

———→ 回復の順

## RTS

ソース型式 RTS

サブルーチンからの復帰命令です．サブルーチンを呼んだプログラムにもどります．コンディション・コードは変化しません．

動作としては，リターン・アドレスをSスタックから回復して，PCに格納します．つまり，スタックに退避されているPCの値を回復すればよいのであり，サブルーチンからの復帰はRTSだけが可能なわけではありません．PULSの項を参照してください．

## SBC

ソース型式 SBCA P；SBCB P

アキュムレータAまたはBからオペランドPで示す内容とボロー(キャリ・ビットCがこれを示す)を引き，結果をアキュムレータに格納します．

コンディション・コードはN，Z，V，Cが影響を受け，Cはボローを示し，キャリの反転値が格納されます．

キャリの反転値がボローとは，2進数の減算では引く数を2の補数に変換し，加算を行うことで減算の結果を得るということからくるものです．

## SEX

| ソース型式　SEX |
|---|

アキュムレータBに格納された2の補数表現である符号付き2進数を16ビットの符号付き2進数に変換して，アキュムレータD(A, Bを連結した16ビット・レジスタ)に格納します．

ロジックのレベルで見た動作としては，アキュムレータBのビット7が1であれば$FFをアキュムレータAに入れ，0であれば$00をアキュムレータAに入れます．

コンディション・コードはN, Zが影響を受けます．

## ST

| ソース型式 | STA　Q；STB　Q |
|---|---|
| | STD　Q；STS　Q |
| | STU　Q；STX　Q |
| | STY　Q |

レジスタの内容を，オペランドQで示すメモリへ書き込みます．ソース型式で示すように，プログラム・カウンタとコンディション・コード・レジスタを除くすべての内部レジスタが対象になります．

8ビット・レジスタはQで示すメモリの1バイトへ，16ビット・レジスタは2バイトの連続するメモリへ書き込まれます．

コンディション・コードはN, Zがデータにより影響を受け，Vはつねにクリアされます．

## SUB

| ソース型式　SUBA　P；SUBB　P |
|---|

ボロー(減算ではキャリ・ビットCがボローを示す)を含まない減算です．

アキュムレータAまたはBからオペランドPで示す内容を引き，結果をアキュムレータに残します．

コンディション・コードはN, Z, V, Cが影響を受けます．Hは不定であり意味をもちません．

SWI

ソース型式　SWI　;SWI2, SWI3

ソフトウェア・インタラプトは6800にも設けられていたSWIのほかに，SWI2, SWI3が追加され，それぞれにベクタ・アドレスが定められています．命令が実行されると，Sレジスタを除くすべての内部レジスタがSスタックに退避され，プログラムの実行は，定められたベクタの示すアドレスに移ります．

退避されたコンディション・コードのEビットはセットされています．すべてのレジスタを退避したことを示すフラグであり，RTIで復帰するときにこのフラグが利用されます．RTIの項を参照してください．

内部レジスタのコンディション・コードは変化しませんが，SWIではIとFフラグがセットされ，ノーマル・インタラプトとファースト・インタラプトがマスクされます．

SWI2, SWI3ではインタラプトはマスクされません．ソフトウェア・インタラプト命令は，プログラムのデバッグやシステム・コール(OSやモニタの機能の一部をユーザ・プログラムからコールして利用する)に使用されます．

ベクタ・アドレスは以下のとおりです．

SWI　$FFFA, FFFB
SWI2　$FFF4, FFF5
SWI3　$FFF2, FFF3

SYNC

ソース型式　SYNC

命令の実行を停止し，割り込みの発生によって次の命令を実行します．すなわち，ソフトウェアとハードウェアの同期を取って，事象の発生に対して最も高速に対応しようとするものです．

この場合の割り込み入力は，割り込み処理の要求ではなく，同期信号の入力として使用されますが，次の条件があります．

(1) 割り込みがマスクされているか．もしくは割り込み入力信号が3マシン・サイクルよりも短い．

(2) 割り込みが許可状態であり．しかも割り込み入力が3マシン・サイクルよりも長い場合は，割り込み処理が起動される．

コンディション・コードは影響を受けません．

**TFR**

| ソース型式　TFR　R1，R2 |
|---|

内部のレジスタ間でデータの転送を行います．

レジスタR1の内容を，レジスタR2に書き込みます．すべてのレジスタについて可能ですが，同一データ長のレジスタ同士に限られます．

コンディション・コードはR2がCCRでなければ影響を受けません．

**TST**

```
ソース型式　TSTA　;TSTB
　　　　　　TST　Q
```

アキュムレータAまたはBあるいはオペランドQで示すメモリ内容をテストし，コンディション・コードを変化させます．

テストとはオペランドから0を引くことで行われ，オペランドであるアキュムレータA，BまたはQで示すメモリ内容は変化しません．

コンディション・コードはN，Zが影響を受け，Vはつねにクリアされます．

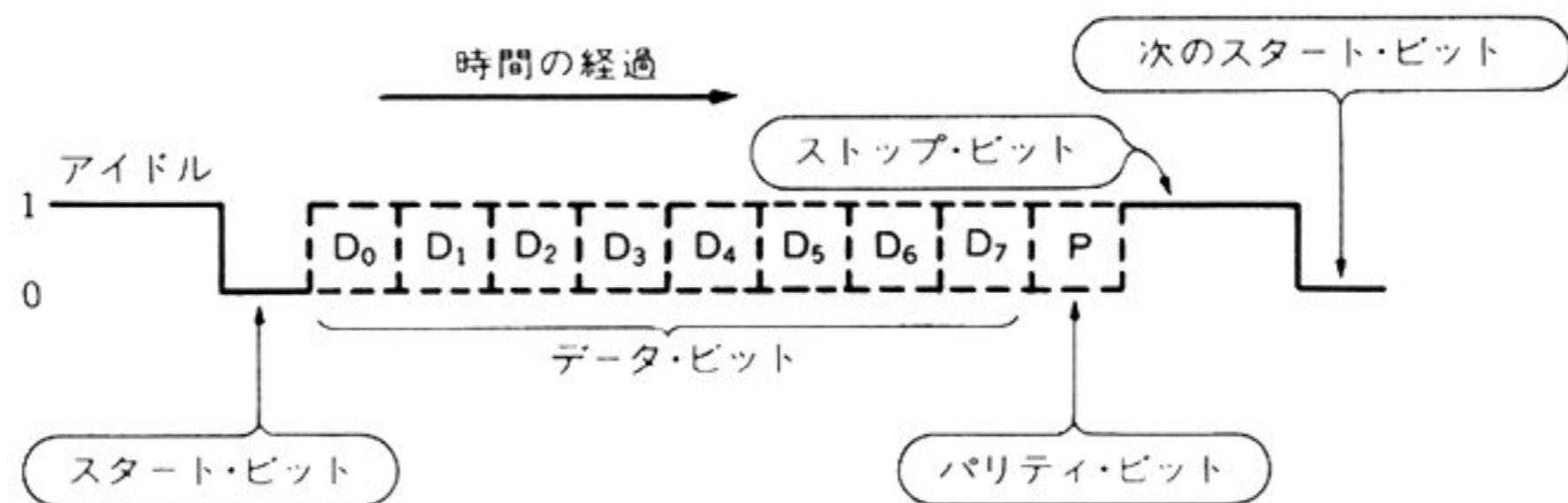
# 第5章
# ペリフェラル駆動のソフトウェア

この章では，第2章で例として示したCPUボードで使用している周辺デバイスを扱うソフトウェアを，プログラム例を示しながら説明します．

ここで取り上げるデバイスは，68系のシステムでは最も一般的なものであり，6809に限らず68000のシステムでも周辺デバイスとして頻繁に使用されます．

マイコン・システムを実際に動かそうとすれば，ペリフェラルを駆動することは，まず最初に覚えなくてはならない基本的なプログラミングであり，OSや言語プロセッサを移植するにしても，避けて通ることはできない部分です．

言い換えれば，ペリフェラルを動かすソフトウェアが組めれば，最低でも実用的なソフトが組めるいっても過言ではないでしょう．

## 5.1 ACIA(6850)によるターミナル入出力

ACIAとはAsynchronous Communications Interfaceの頭文字であり，非同期通信を行うためのI/Oポートです．

非同期通信についてあまり馴染みのない方のために少し説明しておきます．この非同期通信はマイコンに始まったことではなく，テレックスなどで昔から盛んに使用されていました．

電話回線のような通常の通信回線では，8ビットや16ビットといったデータをパラレルで同時に伝送することはできません．そこで，1ビットずつシリアル・コードで伝送しなくてはなりません．

ここで，仮にバイト単位のデータを伝送しているものと考えてください．0と1のディ

図5.1 非同期通信のシリアル信号波形の例

ジタル信号がまったく連続して伝送されているとすれば，受けるほうでは最初からしっかりと送られてくる信号に同期して受信しなければ，どこがバイトの区切りなのかわからなくなってしまいます．

これでは信頼性の高い通信は困難です．同期を使用した同期式シリアル通信というものもありますが，その説明は省略します．

一般の通信回線では，データ信号と同時に，同期信号も送るなどということは困難であったので，私たちの先輩は信号のうまい約束事を作ることで，この問題を解決して実用化してきたのです．

これが，ACIA が行う非同期式シリアル通信なのです．

### ● 信号の波形

非同期通信の信号波形の例を図5.1に示します．

スタート・ビットは次にデータが続くことを示し，時間の長さはデータ・ビットの1ビット分と同じ長さです．この長さを単位時間として1とします．スタート・ビットは0レベルに定められており，受信側ではこの信号で1語分(この場合は8ビット)の同期を取ります．

ストップ・ビットは1レベルであり，データの終了を示します．

ストップ・ビットの長さは，1または2ビット分の長さが使用されますが，この約束とはミニマムの長さを示すものであり，通信の約束としては最大長は定められていません．つまり，1または2ビット以上と理解してください．

昔は1.41ビット長が使われていましたが，これはメカニカル式のテレタイプ通信機を対象としたもので，マイコンのディジタル回路では半端な値なので，マイコンで使われることはほとんどありませんが，この近似値として1.5ビット長は使用されることがあります．

アイドルとは，信号が伝送されないむだな時間の経過を意味します．このレベルは1であり，アイドルとは，ストップ・ビットが無制限に引き延ばされたものと考えることもできます．

パリティ・ビットは，使用する場合と使用しない場合があります．使用する場合には，偶数パリティか奇数パリティかを定めなくてはなりません．

偶数パリティとは，データ・ビットに含まれる1レベルの数が偶数の場合にはパリティ・ビットを1とし，奇数の場合には0にします．

奇数パリティでは，データ・ビットに含まれる1の数が奇数の場合に1，そうでない場合に0とします．

受信側では，このパリティ・ビットを利用して受信のエラー・チェックを行います．1データのビット数は，図の例では8ビットですが，7ビットだけのこともあり，ACIAでは7ビットまたは8ビットを選択することができます．

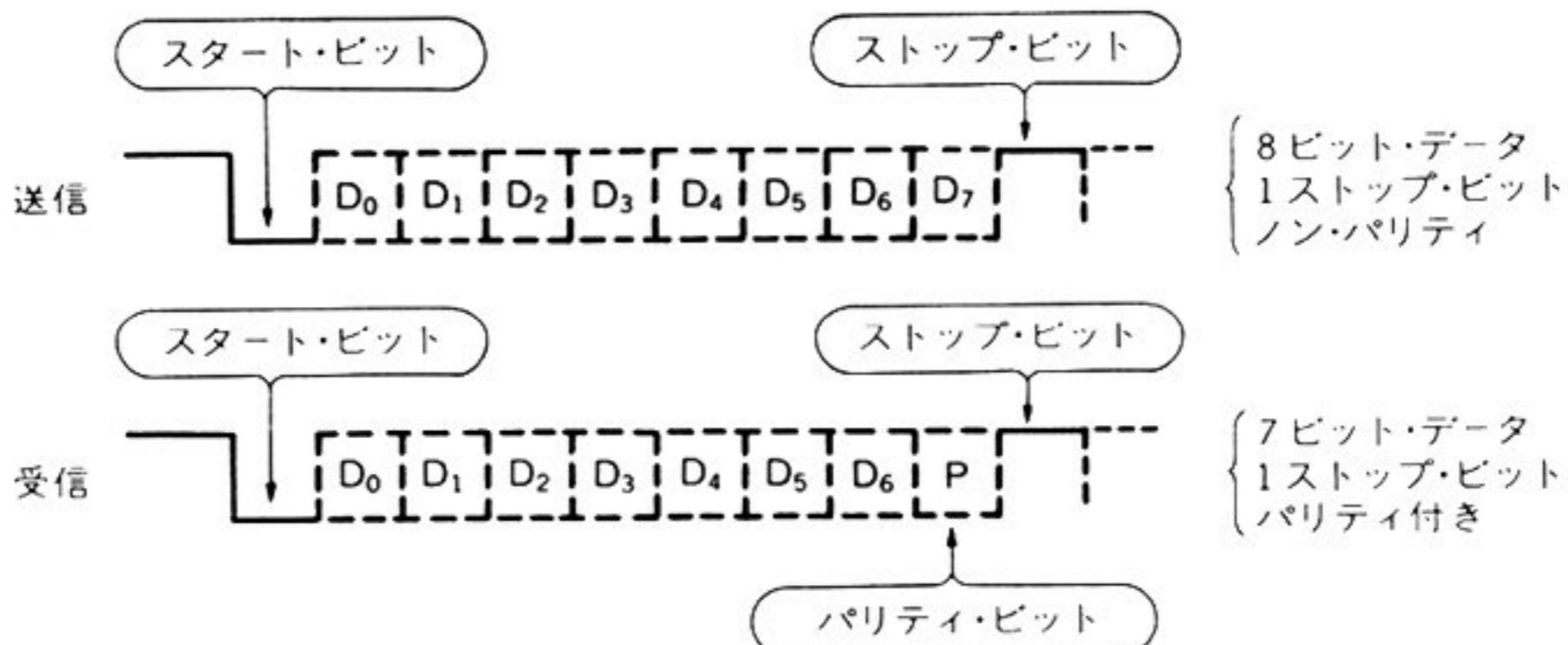
非同期通信で伝送するデータ・コードは文字コードを想定しているので，ASCII文字であれば7ビットで十分なわけです．

### ● ボーレイト

シリアル・コードによる通信のスピードを示すのに，“ボーレイト”と呼ばれる単位を古くから使用しています．

ボーレイトは1秒間に伝送するビット数を示し，その逆数は1ビットの時間を示します．

ボーレイト(Baudrate)とは，フランスの発明家 J.M.E.Baudotから取ったものですが，最近では数値をそのまま示すbps(bit per second)も使用されます．

## ◆ ACIAのイニシャライズ・プログラム

ACIA(6850)の使用を開始するには，どのようなモードで使用するのかを，ACIAに知らせてやらなければなりません，つまりイニシャライズが必要です．

このためにはまず，シリアル・コードの型式について，以下のことを決定しなくてはなりません．

(1) データのビット数，7または8

(2) パリティ・ビットの有無，ありの場合は奇数か偶数か

(3) ストップ・ビットの長さ，1または2

(4) ボーレイト

以上の四つの内容は，使用するターミナルやモデムの仕様に合わせることになりますが，

図5.2 送信と受信で，信号のフォームが異なった場合

互いにプログラマブルなことが多いのです．JISキャラクタを使用する場合には，データ・サイズは8ビットが必要です．

これらの四つの内容は，通信を行う機器間で互いに同じ約束に従うのが原則ですが，必ずしも同じでなければまったく通信できないということでもありません．図5.2では，送信と受信でフォームの異なる例ですが，この場合を考えてみましょう．

受信側のパリティ・ビットには送信のビット7が対応しています．ASCIIコードではこのビットは使用されないので常に0と考えられます．

つまり，パリティ・エラーを頻繁に起こすことになりますが，これは承知しているので無視します．送信と受信が逆になった場合には，ビット7をマスクすればよいことになります．

以上のことを考慮して，第3章の表3.5を参照してください．これを参考にしてイニシャライズのプログラムを作ることになります．

前置きがたいへん長くなってしまいましたが，リスト5.1にイニシャライズ・プログラムの例を示します．

190行では，3をコントロール・レジスタに書き込んでいます．ACIAのリセットであり，ACIAはハードウェアでリセットすることができませんので必要です．このコードは，表3.5のテーブル4を参照してください．

次に$15をコントロール・レジスタに書き込んでいます．$15のビット・パターンである00010101と表3.5のテーブル2，3，4とを見比べてください．

この例では，ACIAを以下の仕様でイニシャライズしていることが読み取れると思います．

リスト5.1 シリアル・インターフェース 6850 ACIA のイニシャライズ

```
00100                          NAM    TERM.I/O
00110                          OPT    M
00120                    **
00130                    **
00140        E010        ACIAC   EQU    $E010
00150        E011        ACIAD   EQU    ACIAC+1
00160                    **
00170 0000 8E    E010 INZACI  LDX    #ACIAC
00180 0003 86    03           LDA    #3       } ACIA をリセット
00190 0005 A7    84           STA    ,X
00200 0007 86    15           LDA    #$15      8ビット＋1ストップ・ビット,
00210 0009 A7    84           STA    ,X        ÷16（クロック）
00220 000B 39                 RTS
00230                    **
```

(1) 語長は8ビット

(2) パリティ・ビットなし

(3) ストップ・ビットは1

(4) クロック・カウンタは÷16

(1)から(3)はすでに説明したとおりです．(4)はボーレイトにも関係します．

(4)の÷16とは，ACIA内部の通信用クロックのカウンタを16分周にセットすることですが，テーブル4で示すようにこの値は，1, 16, 64のいずれかが選択できます．

しかし，非同期通信ではスタート・ビットの受信によって，ACIA内部のカウンタを同期させ，1語のデータ受信に備えるので，この動作ができない÷1では，特別の場合を除いては具合が悪いことになります．従って通常では，÷16もしくは÷64を使用することになります．

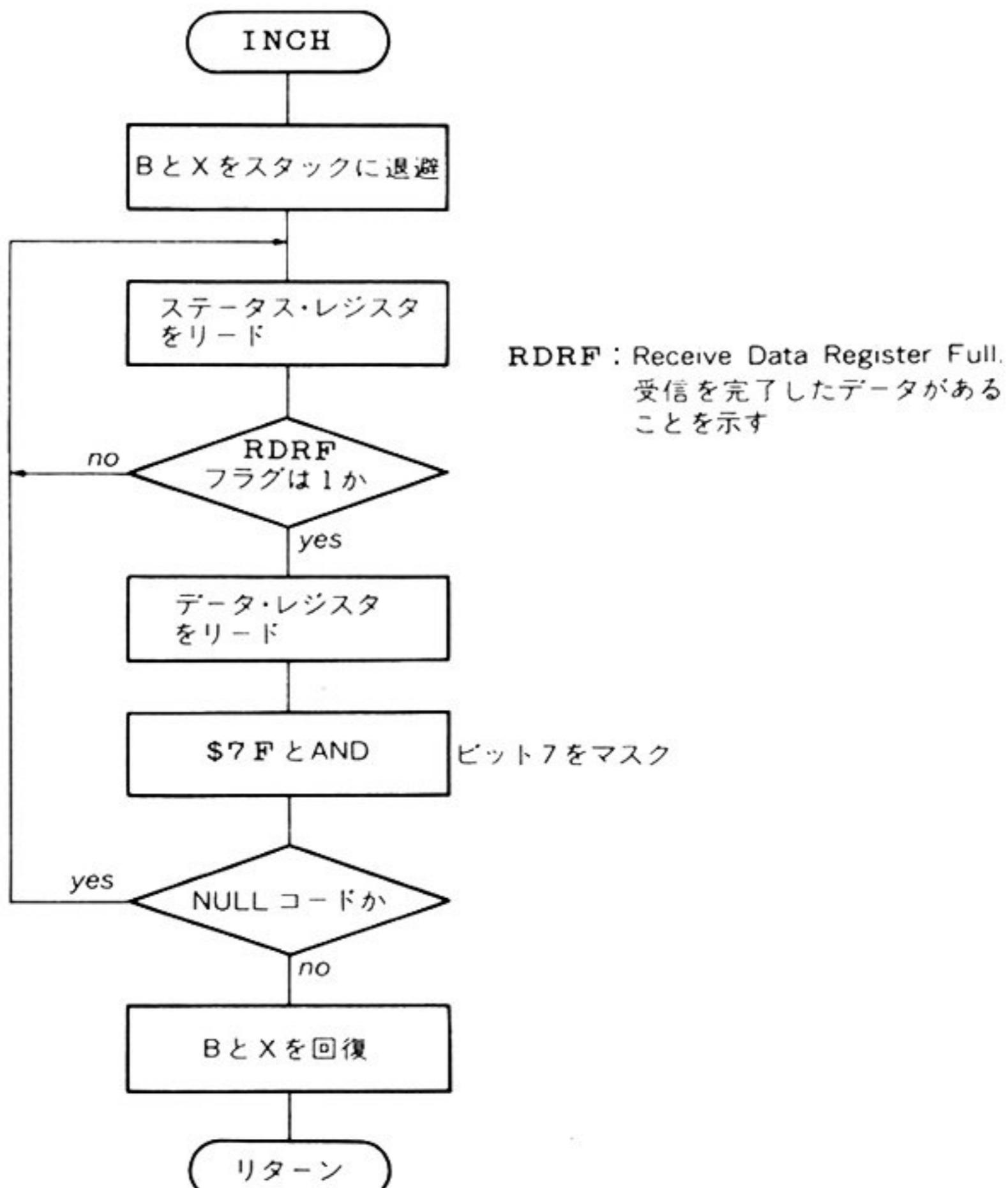
この例では÷16を使用したので，ACIAの送信クロックおよび受信クロックには，ボーレイトの16倍の周波数のクロックを入力することになります．

## ◆ 1文字入力プログラム

ターミナルから1文字を入力するプログラムを**リスト5.2**に紹介しておきます．このプログラムは，以下の仕様で作られています．

(1) 入力したデータはAレジスタに残してリターン

(2) A以外のレジスタは変化しない

(3) キーが打たれるまでリターンしない

リスト5.2　1文字入力

```
00240                         **
00250                         * INPUT ONE CHAR. *
00260 000C 34      14         INCH    PSHS  B,X        BとXをスタックにセーブ
00270 000E 8E      E010               LDX   #ACIAC
00280 0011 E6      84         INCH01  LDB   ,X         ステータス・レジスタをリード
00290 0013 57                         ASRB             } RDRFのステータスをテスト
00300 0014 24      FB                 BCC   INCH01     }
00310 0016 A6      01                 LDA   1,X        データ・レジスタをリード
00320 0018 84      7F                 ANDA  #$7F       ビット7をマスク
00330 001A 27      F5                 BEQ   INCH01     NULLコードならリターンしない
00340 001C 35      94                 PULS  B,X,PC     BとXをスタックから取り出し,同時に
00350                         **                       PCも取り出してリターン
```

図5.3　1文字入力のフローチャート

フローチャートは図5.3です．これを参照しながらリストを見てみましょう．まず260行でレジスタBとXをスタックに退避します．BとXはこのサブルーチン内で使用しますが，サブルーチンからリターンするときには，スタックから再びもどして元の状態にしておき，コールした側から見ればあたかも変化していないように処置します．

次のINCHO1(280行)ではステータス・レジスタをリードして，受信データ・レジスタにデータが用意されたかどうかを見ています．この状態はステータス・レジスタのビット0で示されるので，ビット・テスト命令でテストできますが，この例では右シフト命令を使用して，目的のフラグをキャリ・ビットに入れてからテストしています．

320行は，読み込んだデータのビット7をマスクしています．このプログラムではASCIIコードだけを対象にしたので，フォーマットのミスマッチに対する処置ですが，JISコードも使用する場合は，この行を削除してください．

340行は，BとXレジスタおよびプログラム・カウンタのPCをスタックからもどしています．BとXは退避しておいたレジスタの復帰ですが，PCはサブルーチン・コールによって退避されたもどり番地をプログラム・カウンタにもどしています．つまりこの命令では，RTS命令も同時に含んでいます．

### ◆ バッファ入力プログラム

1行分の文字列を入力するプログラムの例を紹介します．このプログラムの仕様は，次のとおりです．

(1) キャリッジ・リターンが入力されるまで，または80文字になるまで文字列を入力してXで示すバッファに格納する

(2) リターン時のXレジスタは最終文字の次のアドレスを示す

(3) Xレジスタ以外は変化しない

このプログラム例をリスト5.3，フローチャートを図5.4に示します．

このプログラムでは先の1文字入力(INCH)と，エコー・バックのため次に説明する1文字出力(OUTCH)を使用しています．

フローチャートを参照することで，リストの内容は理解できると思いますが，このプログラムではスタックを利用したローカル変数を使用しています．その点について説明しておきます．

400行では，Aレジスタにロードされた80をスタックSにプッシュし，これをスタック領域に割り当てられた変数として，文字数のカウントに使用します．

リスト5.3　バッファ入力

```
00360                    **
00370                    * BUFFER INPUT: X=BUFF. ADR. *
00380 001E 34    02      BUFIN  PSHS  A
00390 0020 86    50             LDA   #80        文字カウント
00400 0022 34    02             PSHS  A
00410 0024 8D    E6      BUFIN2 BSR   INCH
00420 0026 A7    80             STA   ,X+
00430 0028 81    0D             CMPA  #$0D          CR? CRコードか
00440 002A 27    0C             BEQ   BUFFIN
00450 002C 8D    0E             BSR   OUTCH
00460 002E 6A    E4             DEC   ,S         80文字目か
00470 0030 27    02             BEQ   BUFIN1
00480 0032 20    F0             BRA   BUFIN2
00490                    **
00500 0034 86    0D      BUFIN1 LDA   #$0D
00510 0036 A7    80             STA   ,X+
00520 0038 32    61      BUFFIN LEAS  1,S        ローカル変数域を解除
00530 003A 35    82             PULS  A,PC
00540                    **
```

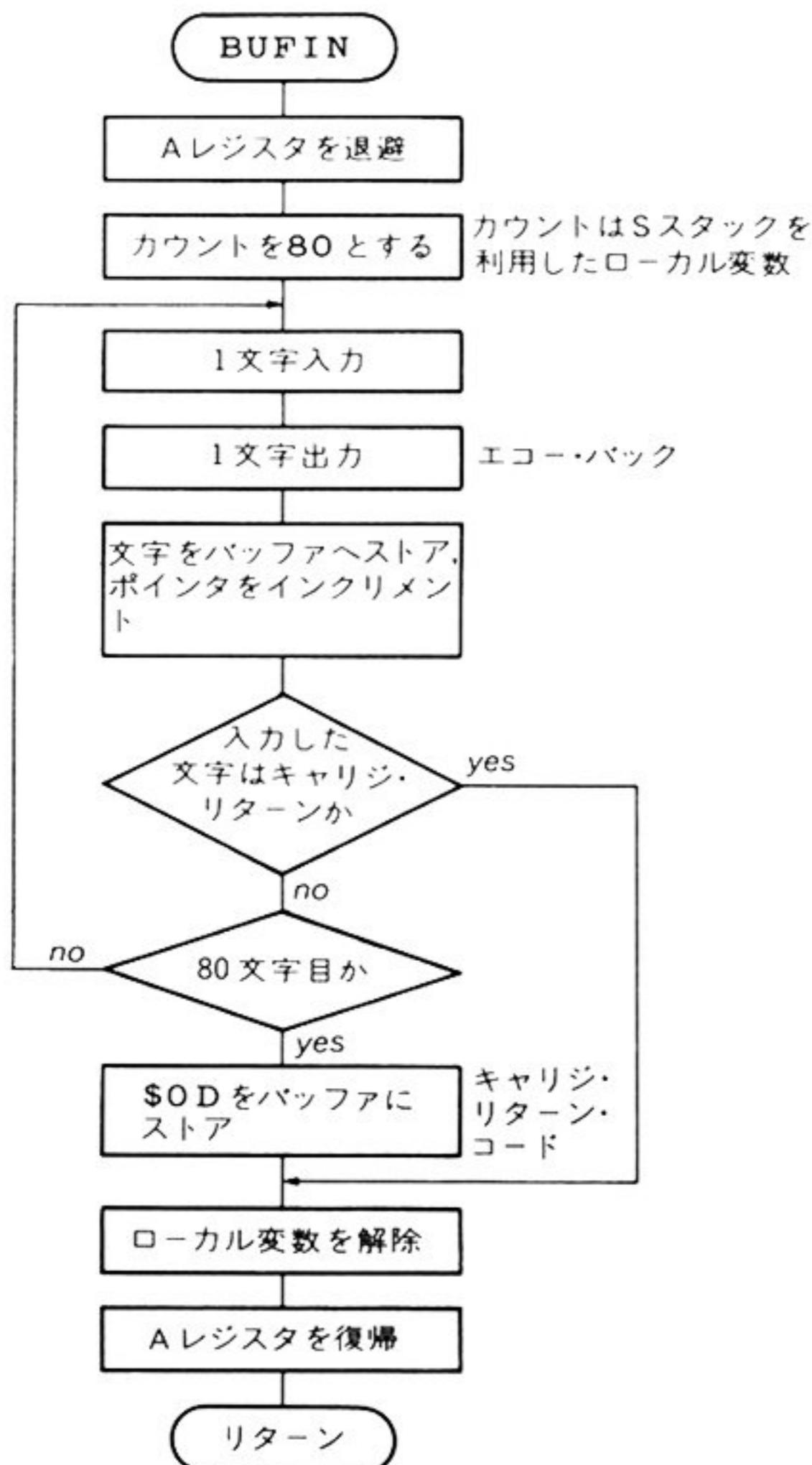



図5.4
バッファ入力のフローチャート

460行ではスタック上のこの変数をデクリメントして，次の行で0かどうかを判断します．

520行では，スタックの不要となった変数領域を，スタック・ポインタを1インクリメントすることで解除しています．

ローカル変数としては最も単純な例ですが，豊富なアドレシング・モードのおかげでこのようなことが容易に実現できるのも6809の特徴です．

## ◆ 1文字出力プログラム

ターミナルへ1文字を出力するプログラムは次の仕様で作りました．

(1) Aレジスタの内容を出力する

(2) すべてのレジスタは変化しない

この仕様は最も単純な例です．昔では，キャリッジ・リターン・コードの後にはnull(00)コードをいくつか自動的に送出するといったことをよくやりました．これは機械印字式の端末が使われていたためで，キャリッジ・リターンでは1文字を印字するよりも余計に時間が必要だったため，その時間を稼ぐためにnullコードが置かれていました．

最近では高速のビデオ・ターミナルが一般的に使用されるので，この必要はほとんどなくなりました．

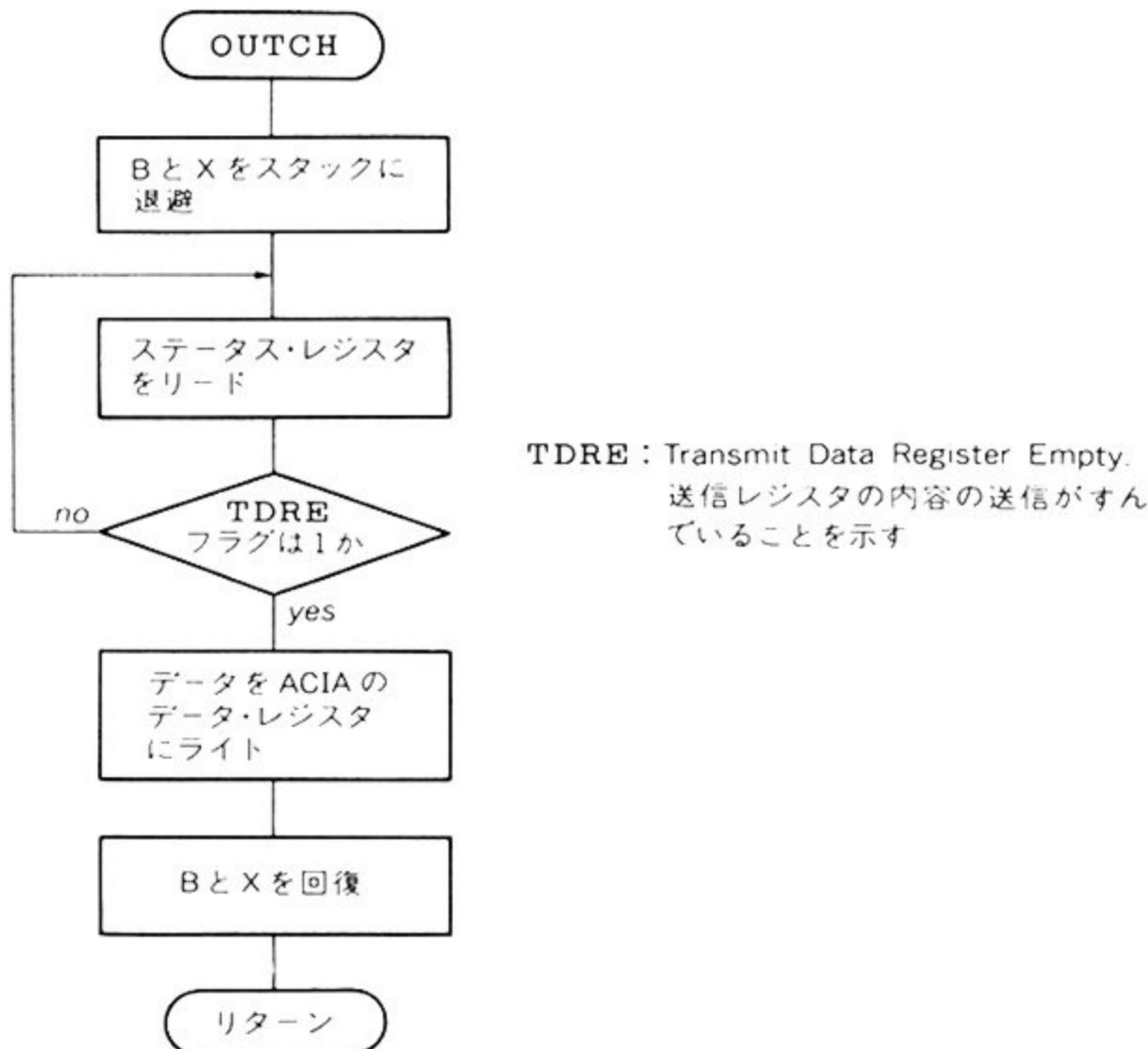
プログラム例を**リスト5.4**に示します．簡単なプログラムであるので，コメントと**図5.5**のフローチャートを参照してこの内容は容易に理解できると思います．

600行では，ステータス・レジスタのビット1が示すTDRE(Transmit Data Register Empty)フラグをテストします．ACIA内部の送信レジスタの内容が空(送信済み)であれば，Aレジスタの内容を書き込んでリターン，空でなければ再度ステータス・レジスタをリードします．

630行ではBとXレジスタの回復と同時にプログラム・カウンタも回復して，ここでもRTS命令を含んでいます．6809のアセンブラ・プログラムでは，サブルーチンの最後が必ずしもRTS命令でないことに注意してください．

## ◆ バッファ出力プログラム

メモリに置かれた文字列をターミナルに出力するプログラムです．1文字の出力は**リスト5.4**のOUTCHを使用しています．

このプログラムの仕様は次のとおりです．

リスト5.4　1文字出力

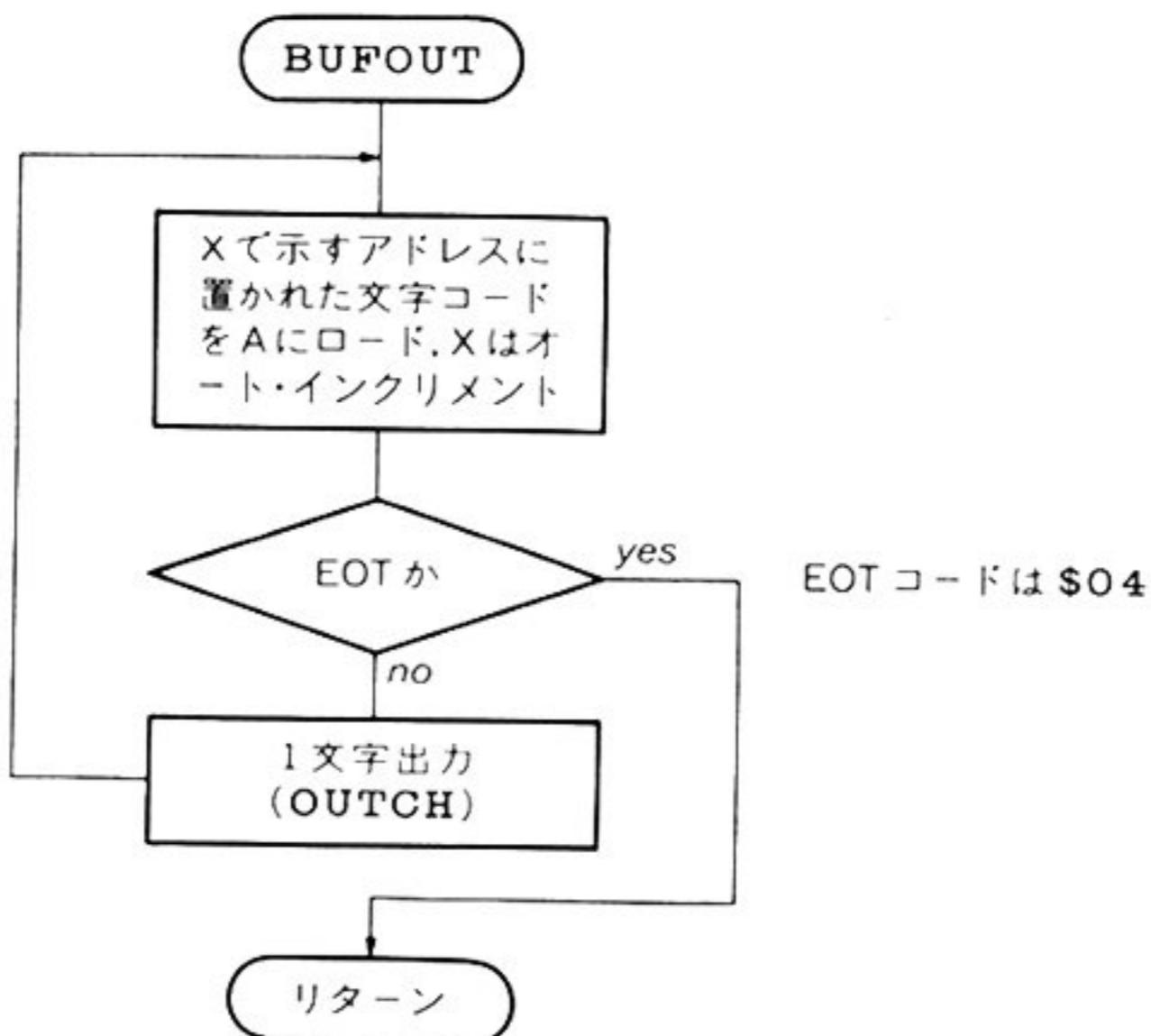
```
00550                       **
00560                       * OUTPUT ONE CHAR. *
00570 003C 34      14       OUTCH  PSHS  B,X       BとXをセーブ
00580 003E 8E      E010            LDX   #ACIAC
00590 0041 E6      84       OUCH1  LDB   ,X        ステータス・レジスタをリード
00600 0043 C5      02              BITB  #2        TDREをテスト
00610 0045 27      FA              BEQ   OUCH1
00620 0047 A7      01              STA   1,X       Aの内容をデータ・レジスタにライト
00630 0049 35      94              PULS  B,X,PC
00640                       **
```

図5.5　1文字出力のフローチャート

(1)　文字列の最後は$04(EOT)とする

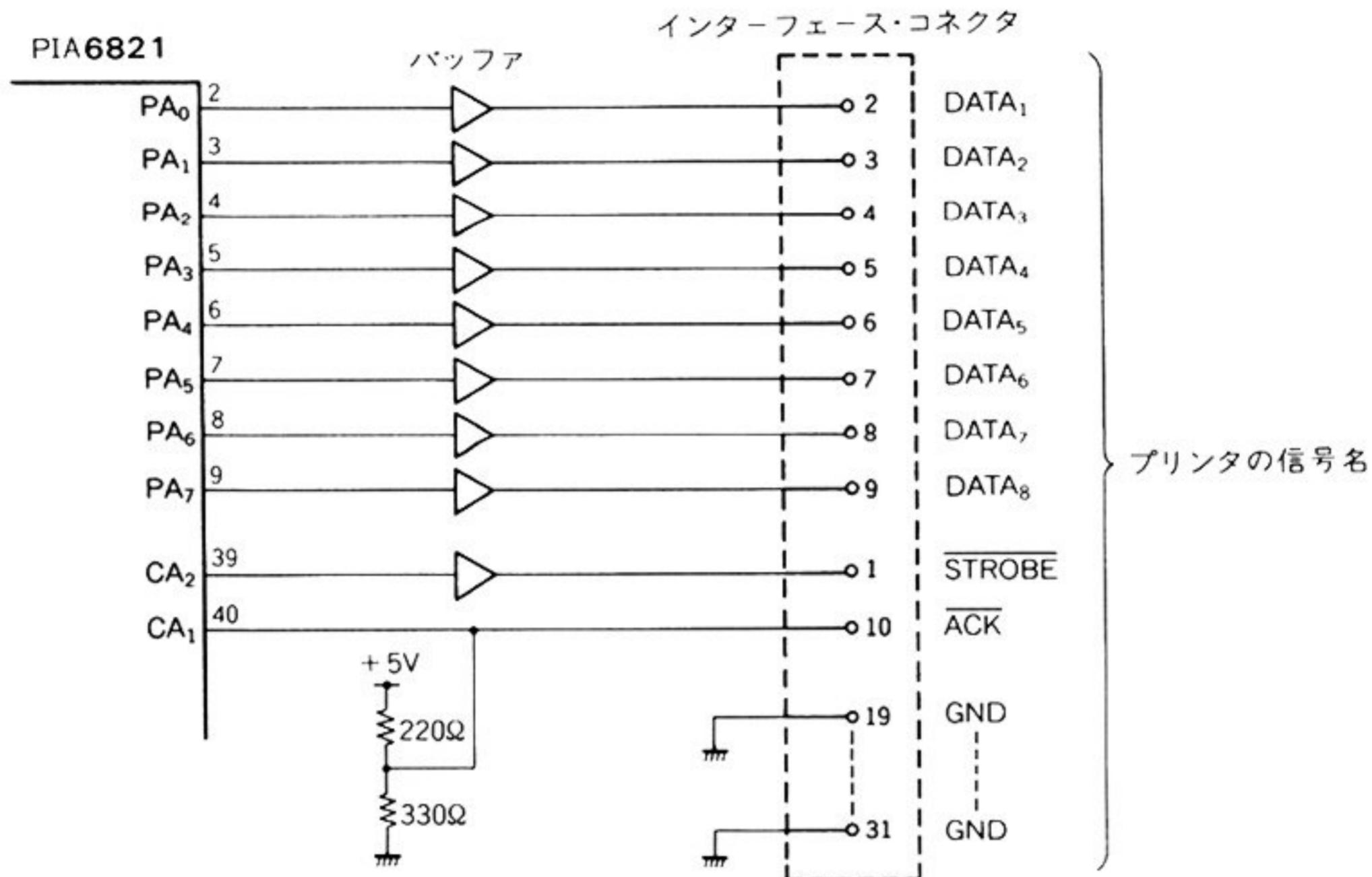
(2)　文字列の先頭アドレスはXレジスタが示す

(3)　AとXレジスタは保存されない，ほかのレジスタは保存される．リターンしたときのXの内容はEOTのコードが置かれた次のアドレス値．Aの内容はEOTコード，すなわち$04

プログラム例をリスト5.5，フローチャートを図5.6に示しました．これも簡単なプログ

リスト5.5 バッファ出力

```
00650                      **
00660                      * BUFFER OUTPUT: X=BUF. ADR. *
00670                      * TERMINATOR=04 *
00680 004B A6   80         BUFOUT LDA    ,X+
00690 004D 81   04                CMPA   #4        EOTか
00700 004F 27   05                BEQ    BUFOT1
00710 0051 17   FFE8              LBSR   OUTCH
00720 0054 20   F5                BRA    BUFOUT
00730 0056 39              BUFOT1 RTS
00740                      **
```

図5.6 バッファ出力のフローチャート

ラムであるので，両者を参照して内容は理解できると思います．

Xレジスタに文字バッファのアドレスをロードしてこのサブルーチンをコールすれば，バッファの文字列がターミナルに出力されます．

文字列の最後にEOTコードを置くことを忘れないでください．これがないと，このプログラムは終了しないことになります．

## 5.2 セントロニクス・スタンダードのプリンタ出力

現在では、私たちがパソコンで使用するプリンタのほとんどが，セントロニクス・イン

図5.7 セントロニクス・プリンタのインターフェース

●バッファは7404, 7417などのオープン・コレクタ出力のタイプのものを使うべきだが，普通のTTL出力でもほとんどの場合問題ない．

ターフェースの仕様にしたがったものであり，基本的には機種やメーカを問わず，同一のインターフェース，同一のプログラムを使用することができます．

ここではPIA(6821)を使用した，最も簡単な例を紹介します．これでも十分に実用的であり，筆者もこのままで使用しています．

セントロニクス・スタンダードのインターフェースでは，8本のデータ線を使用したパラレル転送です．

タイミング用にデータ・ストローブ，ビジィ，アクノレッジがあり，ハンドシェイクを行います．このほかにも，プリンタの初期化やエラーを示す信号がありますが，必ずしもこれらを使用したインターフェースにしなければならないわけではありません．

インターフェースの回路例を図5.7に示しておきます．コネクタも36ピンのアンフェノール・コネクタが使用され，信号とピン・ナンバの対応も統一されているので，このままではほとんどのプリンタに使用できるはずです．

この例では，ビジィ信号は省略して，ストローブとアクノレッジだけでハンドシェイクを行っています．

リスト5.6 プリンタ出力ポートのイニシャライズ

```
00750                        **
00760                        * INITIALYZE PRINTER PORT *
00770           E02C         LSTPTA EQU    $E02C
00780           E02D         LSTPTC EQU    LSTPTA+1
00790                        **
00800                        **
00810 0057 7F        E02D LSTINZ CLR    LSTPTC    コントロール・レジスタをクリア
00820 005A CC        FF2E        LDD    #$FF2E    イニシャライズのコード
00830 005D FD        E02C        STD    LSTPTA
00840 0060 39                    RTS
00850                        **
```

### ◆ プリンタ出力のイニシャライズ・プログラム

6821の使用を開始する場合にも，イニシャライズが必要です．このプログラムを**リスト5.6**に示します．イニシャライズで書き込む6821のレジスタの機能については，第3章の**表3.3**を参照してください．

810行ではコントロール・レジスタをクリアしています．6821では，$\overline{\mathrm{RESET}}$信号があり，電源投入時にハード的に内部レジスタがクリアされるので，この行の命令は必ずしも必要ではありませんが，ない場合にはイニシャライズ・プログラムが2回以上実行された場合に支障が生じます．

すなわち，**表3.3**のテーブル1が示すように，コントロール・レジスタのビット2によって，書き込まれたデータは出力データなのか方向レジスタへの書き込みなのかを区別します．使用中はこのビットを1にして入出力レジスタをアクセスするわけですが，イニシャライズでは方向レジスタの決定のために，このビットをクリアしなくてはならないからです．

次の820行と830行では，イニシャライズのコードを書き込んでいますが，方向レジスタとコントロール・レジスタへの書き込みを，16ビットのストア命令を利用して一度に行っています．

上位の$FFは方向レジスタに書き込まれ，Aポートの方向をすべてのビットについて出力にしています．下位の$2Eは，コントロール・レジスタに書き込まれます．このコードの内容については，**表3.3**のテーブル3とテーブル6を参照してください．

このコードのビット・パターンは，$CA_2$へ101, $CA_1$へ10が書かれます．

リスト5.7　プリンタ1文字出力

```
00860                    **
00870                    * OUTPUT ONE CHAR. TO PRINTER *
00880 0061 B7      E02C PRNTEE STA    LSTPTA        データを書き込む
00890 0064 B6      E02C        LDA    LSTPTA        Dummy read  ダミー・リード
00900 0067 7D      E02D PRNT1  TST    LSTPTC        アクノレッジをテスト
00910 006A 2A      FB          BPL    PRNT1
00920 006C 39                  RTS
00930                    **
00940                    **
```

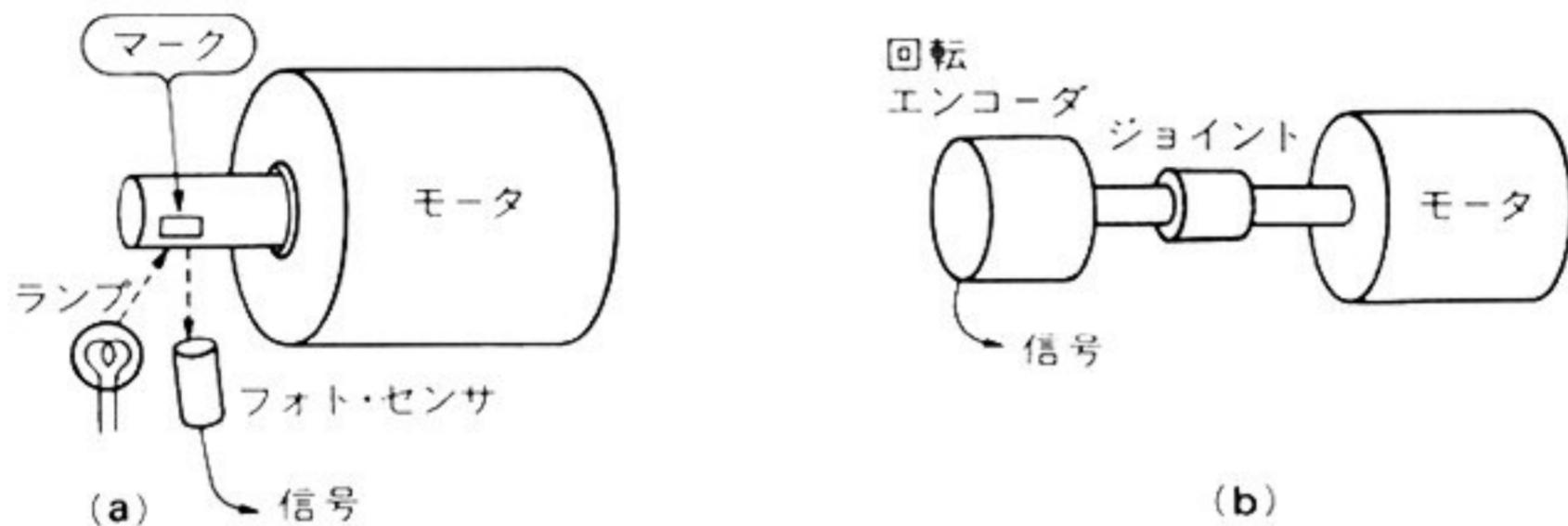
## ◆ 1文字出力プログラム

プリンタへの1文字出力プログラムの例を**リスト5.7**に示します．

プログラムの仕様としては，出力先がプリンタになっただけで，ほかはターミナルの1文字出力と同様です．

880行のPRNTEEの2行目(890行)では，出力ポートを読み込んでいますが，これはストローブを出力するためです．**表3.3**のテーブル6を参照してください．

$CA_2$からは，Aポートのリード動作によってパルスが出力されます．このため読む必要のないデータを読むことになるのです．これをダミー・リードと呼んでいます．

900行のPRNT1では，アクノレッジを確認するために，コントロール・レジスタに対してTST命令を実行しています．

$CA_1$の入力(アクノレッジ)は，コントロール・レジスタのビット7に反映するため，正か負かの判断ですみます．従ってこの判断は次の行にあるように，BPL命令が行います．

## ◆ バッファ出力プログラム

ターミナルのバッファ出力と同様に，メモリ上の文字列を出力するプログラムを考えてみましょう．出力先が異なること以外は，ターミナルのバッファ出力と同じにします．

こうすれば，**リスト5.5**のプログラムがほとんどそのまま使えることに気が付かれると思います．つまり，**リスト5.5**のLBSR　OUCHをLBSR　PRNTEEに書き換えればおしまいです．アセンブラのソース・プログラムとする場合にはラベル名を変えてください．

図5.8 モータの回転検出方法

## 5.3 プログラマブル・タイマ(6840)の応用例

(モータ回転数の計測)

第2章で紹介したCPUボードには，プログラマブル・タイマ(6840)も組み込まれています．これの応用例を紹介します．

具体的な例として，モータ回転数の計測を取り上げました．この例は，モータの生産工場で検査装置として，実際に稼動しているシステムの一部です．

まず，モータの回転を電気信号に変換しなくてはなりません．図5.8にその例を示します．どちらも実績のあるものです．

図(a)はモータのシャフトに反射テープを張り，ランプの反射光をフォト・センサで検出してモータの回転をパルス信号に変換しようとするものです．この特長は，モータと非接触で検出できることです．

図(b)の例は，モータにエンコーダをつなぎ込み，回転をエンコーダでパルス信号に変換します．

今回の場合は，ワウ・フラッタ(回転速度のむら，フラッタはワウよりもむらの周期が短い)の計測もあったので，(b)を採用し，1回転で500パルスを出力するエンコーダを使用しました．

### ◆ 6840の使い方

6840の詳細については，第3章の6840の項を参照してください．この回転計測の例では，6840を図5.9に示すような使い方をしています．

図5.9　回転計測のブロック・ダイヤグラム

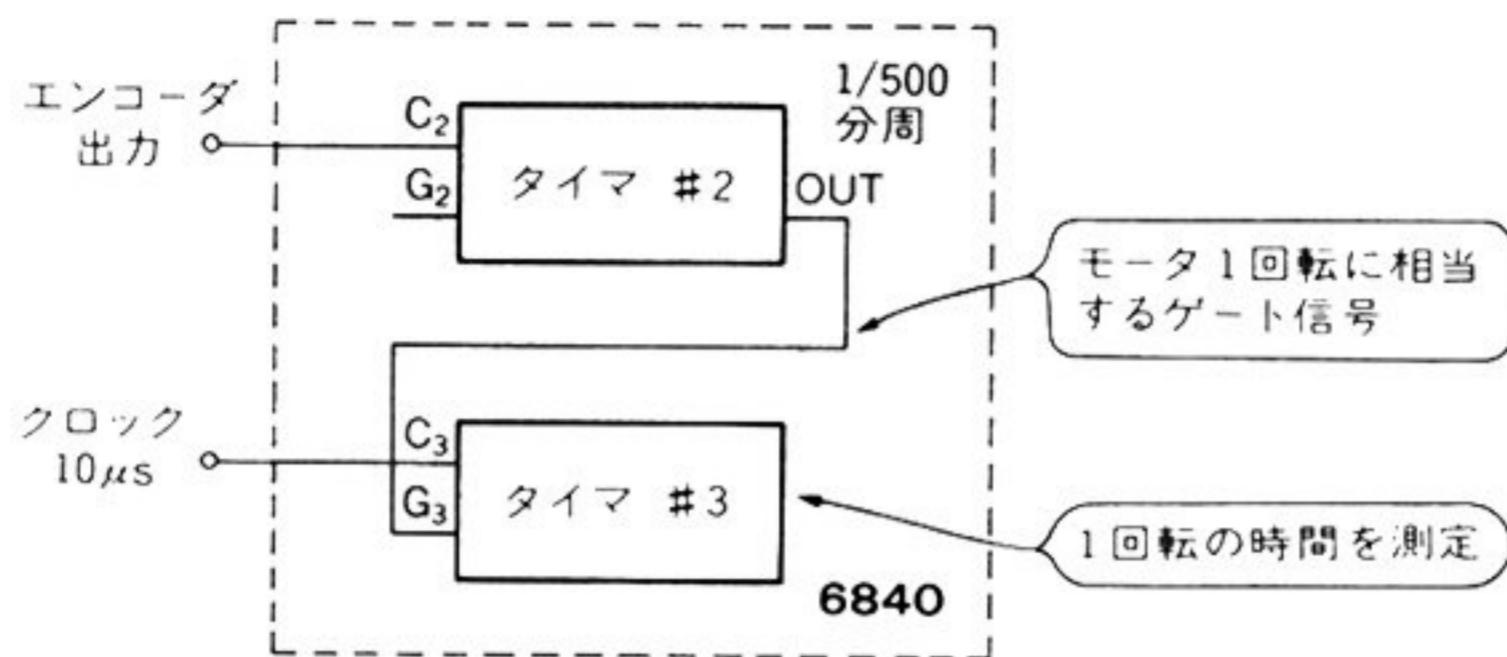
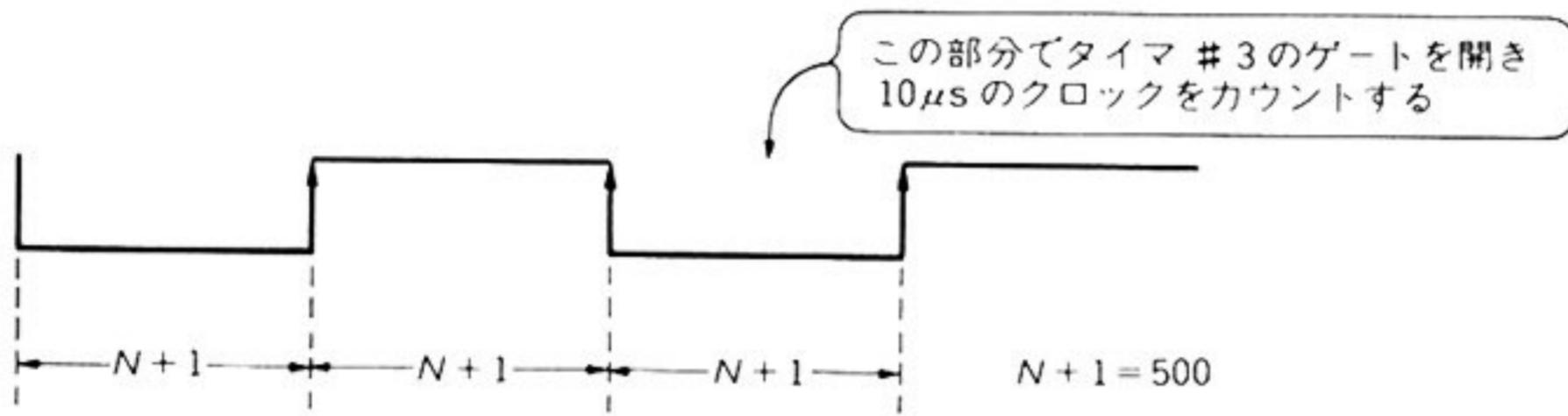
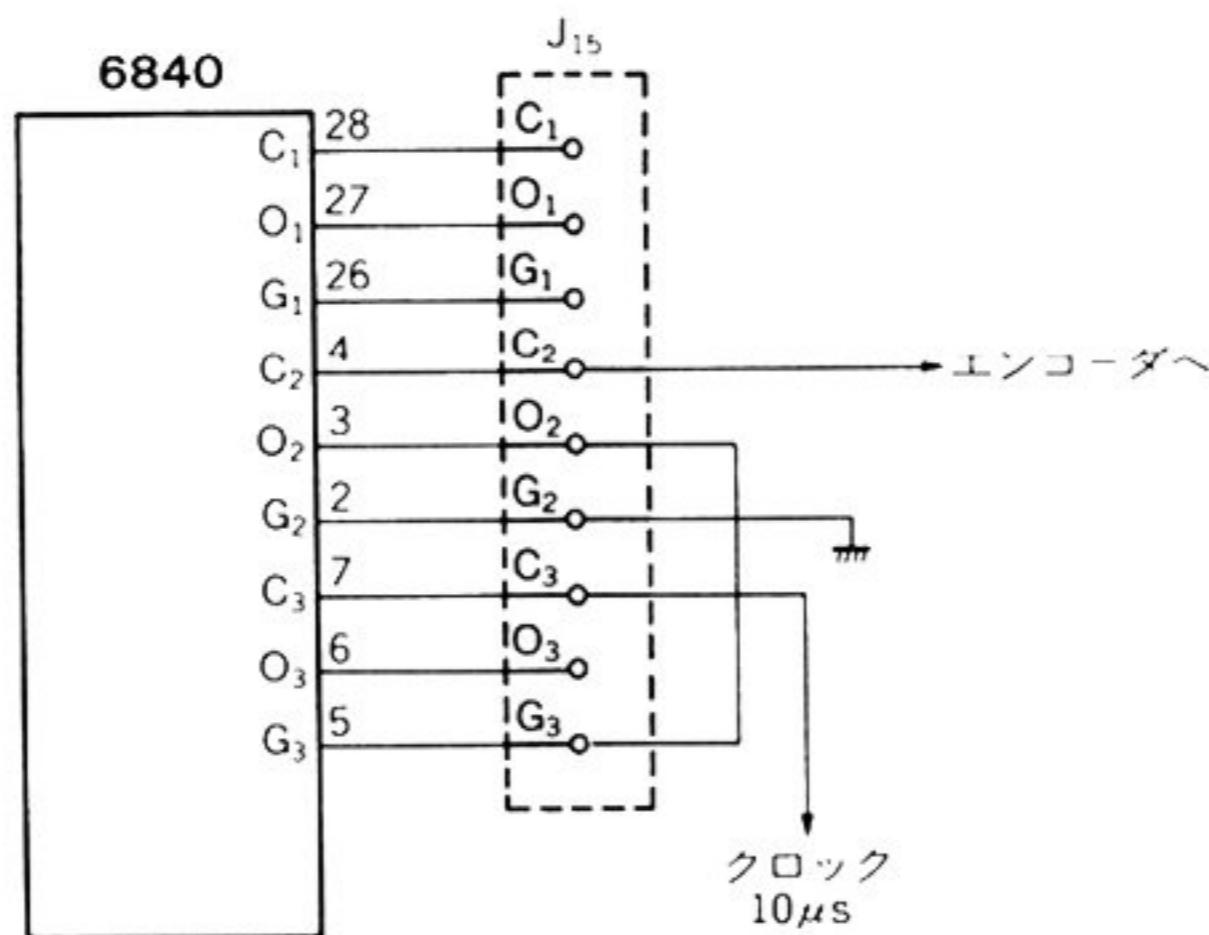



図5.10　6840 タイマ#2の出力

6840では三つのタイマを独立して使用できますが，ここでは#2と#3のタイマを使用しています．

タイマ#2は，エンコーダからの信号を500カウントして図5.10に示す信号を出力します．つまり，モータの1回転の時間に相当するゲート信号を作ります．

タイマ#3では，タイマ#2で作られたゲート信号の時間内で，10 μsのクロックをカウントします．

すなわち，タイマ#3には，モータの1回転に要した時間が10 μsの単位で残ることになり，この逆数を取れば毎秒の回転数が得られるわけです．

カウンタは，どちらもノーマル16ビット・カウンティング・モードを使用します．

第2章で紹介したCPUボードでは，ジャンパ・セレクトの$J_{15}$は図5.11のように配線します．クロックの10 μsはボード内のクロック回路から得ることができ，$J_2$の10 μsに配線します．

この構成で計測できる範囲は，タイマ#3のフルスケールが$FFFF×10 μs＝約0.65

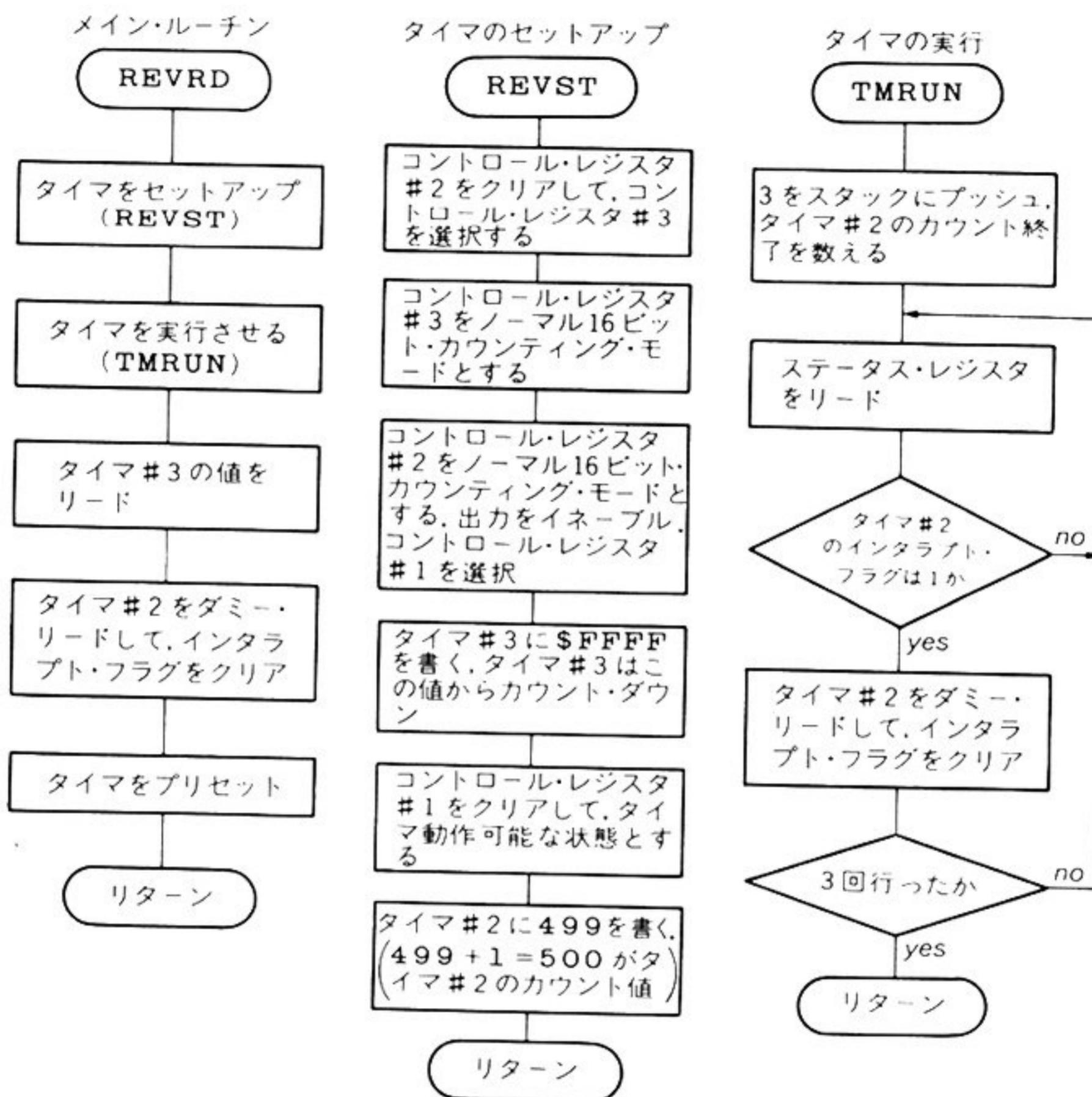
図5.11 回転計測のためのCPUボードのジャンパ・セレクトの設定

秒であるので，92 rpm以上ということになります．

計測値の有効桁数は，タイマ#3のカウント値の桁数で決まるので，フルスケールに近いカウント値を使用できたほうが有利です．実際に応用する場合には，計測するモータの回転数とセンシング方法を考慮して，タイマ#2のカウント値とクロックの周期を決定してください．

ここではタイマ#2を500カウント，タイマ#3のクロックを10 μsとしています．

6840のカウンタはダウン・カウンタです．フルスケール値からタイマ#3のリード値を引いたものが，回転周期になることを注意してください．

### ◆ 回転計測のプログラム

プログラムをリスト5.8，このフローチャートを図5.12に示しておきました．

このプログラムは二つのサブルーチンを含んでおり，リストでは1290行からのREVRDがメインです．REVRDもサブルーチンの形になっているので，このアドレスへサブルーチン・コールすれば，Dレジスタにタイマ#3の値をもってリターンします．

REVRDの使用するサブルーチン，REVSTでは6840の必要な設定はすべて行うので，REVRDを実行する以前に，ほかのプログラムによって6840のイニシャライズを行う必要はありません．

このプログラムの実行によって得られるDレジスタの値は，タイマ#3の初期値である

図5.12 回転計測のフローチャート[()内はサブルーチン名]

$FFFF をダウン・カウントした結果です．従って，この値を RD とすれば，以下の演算によって 0.1 rpm を単位とした回転数が得られます．

60,000,000/($FFFF−RD)

この計算は，第8章で紹介する32ビットの除算ルーチンを使用して行えます．

実際の計測でこのプログラムを実行させる場合には，回転むらがあることを考慮しなくてはならないので，数回の計測を行い，その平均値を求めるべきでしょう．

リスト5.8 回転数の計測

```
00950                     **
00960                     * REVOLUTION MESS. *
00970                     * TIMER ADDRESS
00980      E008           TMC13  EQU   $E008
00990      E009           TMC2   EQU   TMC13+1
01000      E00A           TMR1   EQU   TMC13+2
01010      E00C           TMR2   EQU   TMR1+2
01020      E00E           TMR3   EQU   TMR2+2
01030                     **
01040                     * SET-UP TIMER *
01050 006D 7F    E009 REVST  CLR   TMC2     コントロール・レジスタ#3を選択するため
01060 0070 86    10          LDA   #$10     #2をクリア．表3.7参照
01070 0072 B7    E008        STA   TMC13
01080 0075 86    81          LDA   #$81
01090 0077 B7    E009        STA   TMC2
01100 007A CC    FFFF        LDD   #$FFFF
01110 007D FD    E00E        STD   TMR3     タイマ#3に$FFFF
01120 0080 7F    E008        CLR   TMC13
01130 0083 CC    01F3        LDD   #499
01140 0086 FD    E00C        STD   TMR2     タイマ#2に499（500−1）
01150 0089 39                RTS
01160                     **
01170                     * TIMER RUN *
01180 008A 86    03   TMRUN  LDA   #3
01190 008C 34    02          PSHS  A        タイマ#2の0をカウントする初期値
01200 008E B6    E009 TM21   LDA   TMC2
01210 0091 85    02          BITA  #2
01220 0093 27    F9          BEQ   TM21
01230 0095 FC    E00C        LDD   TMR2     タイマ#2のダミー・リード
01240 0098 6A    E4          DEC   ,S
01250 009A 26    F2          BNE   TM21
01260 009C 35    82          PULS  A,PC     PULL Aは意味がない．LEAS 1，Sと
01270                     **                RTSを兼ねただけ
01280                     * READ REVOLUTION *
01290 009E 8D    CD   REVRD  BSR   REVST
01300 00A0 8D    E8          BSR   TMRUN
01310 00A2 FC    E00E        LDD   TMR3
01320 00A5 34    06          PSHS  D        タイマ#3の値をスタックにセーブ
01330 00A7 7D    E009        TST   TMC2
01340 00AA FC    E00C        LDD   TMR2     タイマ#2をダミー・リードして，
01350 00AD 86    01          LDA   #1       インタラプト・フラグをクリア
01360 00AF B7    E009        STA   TMC2
01370 00B2 B7    E008        STA   TMC13    タイマをプリセット
01380 00B5 35    86          PULS  D,PC     タイマ#3の値をDレジスタに回復して
01390                     **                リターン
01400                     **
```

## ローカル変数とグローバル変数

ローカル変数とは，プログラムの実行中に常に存在するのでなく，特定のサブルーチンが実行されたときのみ存在して，サブルーチンの実行終了と同時に消滅する変数をいいます．

このようなローカル変数は，サブルーチン自身によって生成と解除が行われ，本文で例として述べたように，システム・スタックを利用して容易に実現できます．

サブルーチンの内部だけで使用する変数では，このようなローカル変数とすれば，必要な時にだけその領域が割り当てられるので，メモリの利用効率からも望ましいわけです．

さらに，ローカル変数で処理を行うとは，多重処理において，同じ時間帯に複数のプログラムから呼び出し可能な構造になるというような重要な問題も含んでいます．

これに対してグローバル変数とは，プログラムの実行中は常に存在して，メイン・ルーチンやどのサブルーチンからでも読み書き可能な変数をいいます．

パソコンで使用されるBASICの変数とは，この区別によればグローバル変数ということになります(大型の一部のBASICではローカル変数を定義可能なものもある)．

PASCALやC言語のような関数型の言語では，関数の内部で宣言された変数は一般的にローカル変数であり，関数の内部でのみ有効なことから，別の関数内での変数と同じ変数名を使用しても支障がなく，プログラム開発の効率向上に役立っています．

# 第6章
# 6809の割り込み

## 6.1 割り込み信号

ほとんどの読者の方は，マイクロプロセッサの割り込みについての概念をお持ちのことと思いますが，馴染みの薄い方のために少し説明しておきます．

6809の割り込みについては，ハードウェアによる割り込みとソフトウェア割り込みに大別できますが，まず，ハードウェアによる割り込みについてお話します．今後もことわりなしに割り込みといえば，ハードウェアによる割り込みと解釈してください．

割り込みの基礎的な概念では，実行中のプログラムとは直接には関係のない別の事象が発生し，プロセッサが一時的に実行中のプログラムを中断してその事象のための作業を行い，それが終了したら，何事もなかったかのように，元のプログラムを実行するといったことです．

図6.1 割り込みの概念

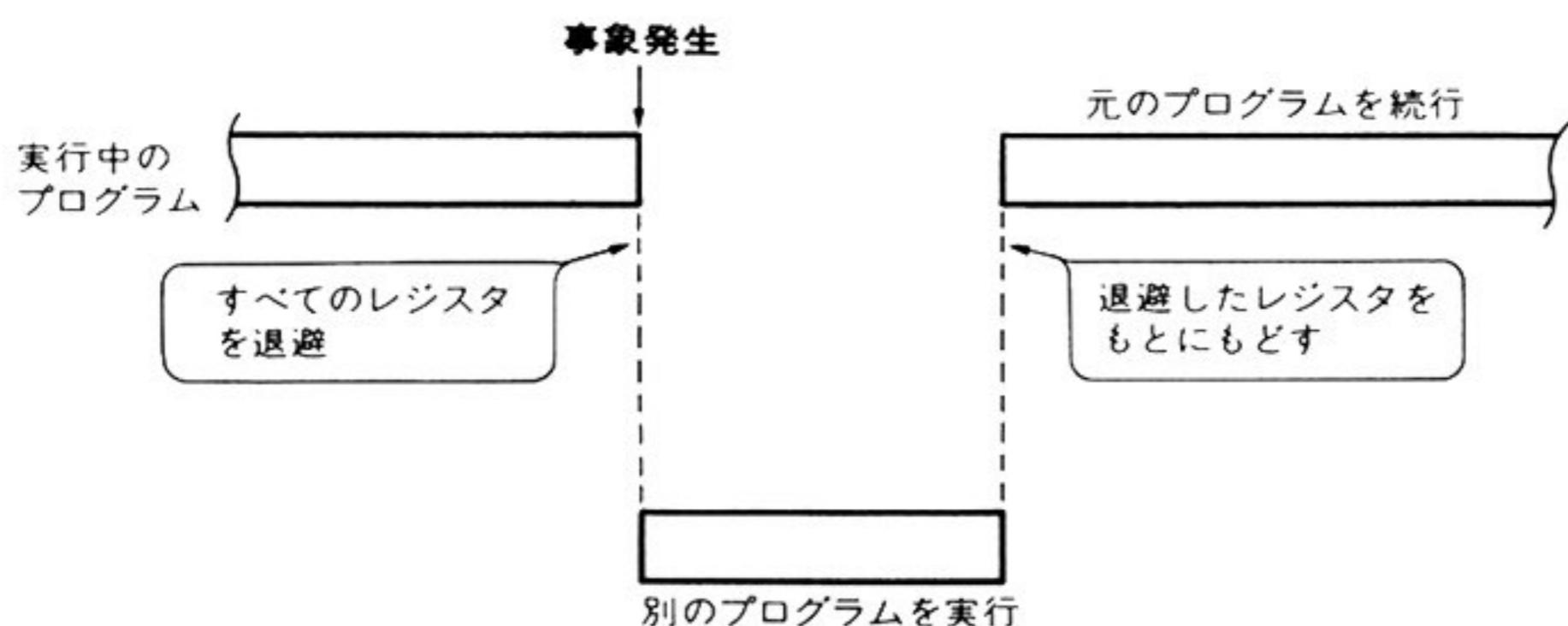
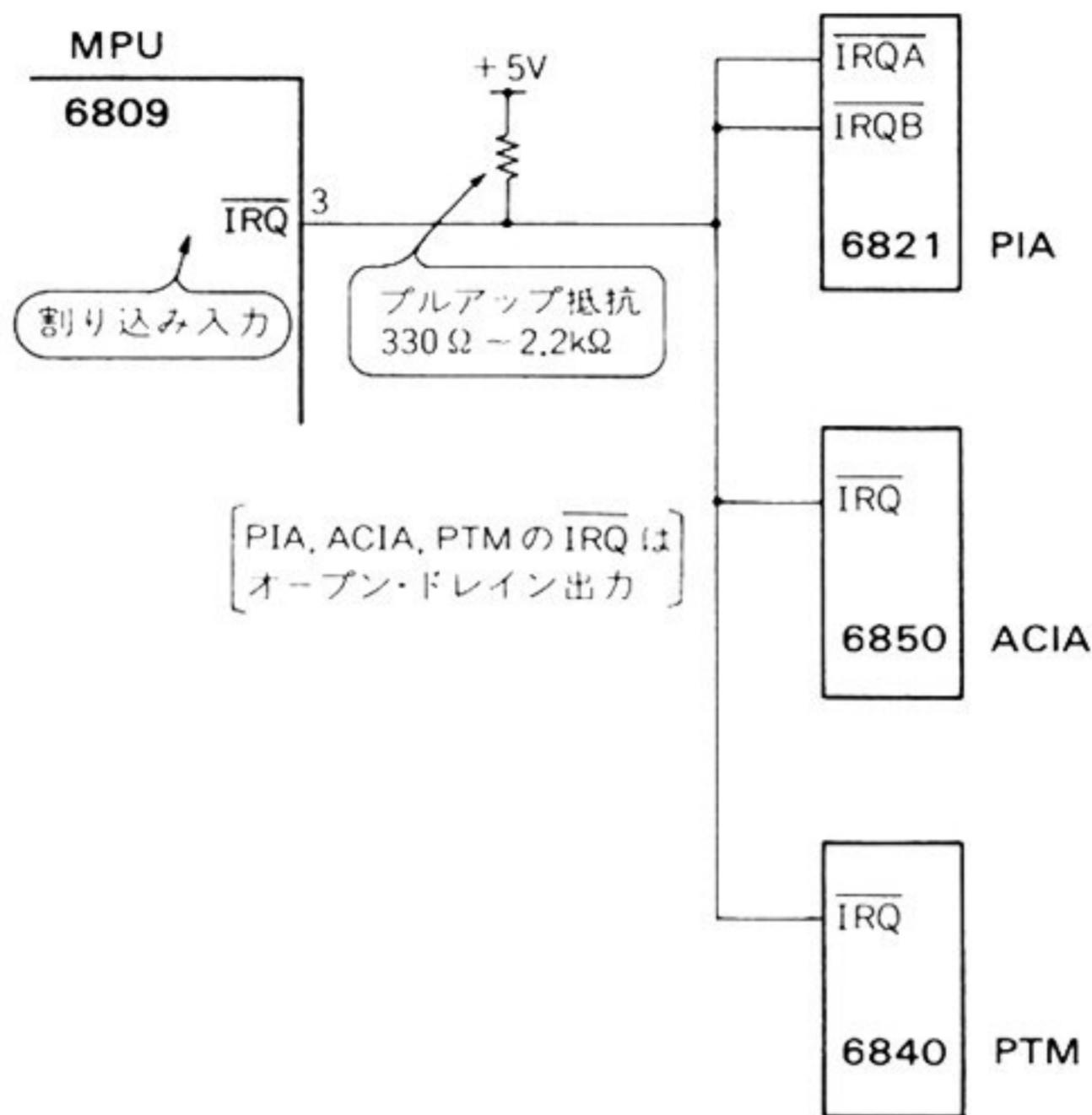



図6.2　割り込みの入力回路

この様子を図6.1に示しました．この図の示すように，プロセッサのレジスタをすべて退避し，再び元にもどせば，実行中のプログラムから見れば空白の時間が発生しただけに過ぎず，この空白の時間がとくに問題とならない限り，何も影響されないことになります．

ここで，一時的に実行されたプログラムを割り込みサービス・ルーチンと呼びます．

このような外部の事象をハード的に入力する信号端子として，6809では$\overline{NMI}$, $\overline{IRQ}$, $\overline{FIRQ}$の3本があります．これらの信号については第2章の「信号と機能の説明」を参照してください．ここでは，一般的な割り込みの使い方について触れておきます．

割り込みの入力回路は，ワイヤードOR接続にするのが最も簡単で一般的です．図6.2は$\overline{IRQ}$の場合の入力例ですが，$\overline{NMI}$や$\overline{FIRQ}$入力についても同様です．

周辺デバイス割り込み要求出力は，オープン・ドレインまたはオープン・コレクタとなっています．

割り込みの要求は，この出力を“L”レベルにすることで行いますが，出力の解除はMPUが要求を発行したデバイスをリードすることで行います．割り込み要求を出すことのある周辺回路を設計する場合にも，出力解除の方法は，同様にするのがよいでしょう．

● $\overline{NMI}$

この割り込みは，$\overline{IRQ}$, $\overline{FIRQ}$とは性格を異にすると考えたほうがよいでしょう．割り込

みの優先順位が最も高く，重要なことは，ソフトウェアでマスク(要求を受け付けない状態)不能ということです．

このことから，通常のユーザ・プログラムで$\overline{\text{NMI}}$が使用されることはあまりありません．システムの異常な状態に対処するためなどに残しておくべきでしょう．

異常な状態とは，メカトロニクスではXYテーブルなどのオーバラン・トラブルにより，異常を知らせるセンサが働いた，または，非常停止ボタンが押されたなどの場合です．この場合には，プログラムの暴走により，割り込みサービス・ルーチンも実行できないことも予想しなくてはならないので，$\overline{\text{NMI}}$による処理だけでは不十分ですが．

別の例では，パワー・ダウン・シーケンスに$\overline{\text{NMI}}$を使用することもあります．

プログラムの実行中に電源が切られた場合，どうしても行わなくてはならない手順がある場合です．この場合には，1次電源(ACライン)の断を検出して$\overline{\text{NMI}}$入力として，2次電圧(+5Vなどの DCライン)が降下するするまでの短い時間に，$\overline{\text{NMI}}$の割り込みサービス・ルーチンで処理してしてしまおうというものです．

これ以外にも応用例はたくさんありますが，いずれにしてもシステムの最も重大な問題に対処するために使用するのが普通です．

● $\overline{\text{IRQ}}$, $\overline{\text{FIRQ}}$

この二つの割り込みが，一般的な割り込み処理として使用されます．

$\overline{\text{IRQ}}$は$\overline{\text{FIRQ}}$よりも優先順位の低い割り込みですが，両者の重要な相違は，割り込み要求に対する応答時間の差にあります．

$\overline{\text{IRQ}}$の要求が受け入れられた場合には，実行中の命令が終了後，すべてのレジスタをスタックに退避して，ベクタのフェッチを行い，サービス・ルーチンの実行を開始します．この間に要する時間は，実行中の命令にもよりますが，ミニマムで20マシン・サイクル(1MHzクロックでは20μs)ですが，$\overline{\text{FIRQ}}$ではプログラム・カウンタとコンディション・コードしか退避しないため，その分だけ時間が短く，ミニマムでは11マシン・サイクルですみます．

$\overline{\text{IRQ}}$と$\overline{\text{FIRQ}}$の使い分けでは，このことを問題にしなくてはなりません．とくに高速の応答が必要でない割り込みでは，$\overline{\text{IRQ}}$を使用すべきです．レジスタの退避および割り込みサービス・ルーチンからのリターン(RTI命令)でのレジスタの回復が自動的に行われるので，プログラムはその分だけバカチョンですむわけです．

$\overline{\text{FIRQ}}$では，割り込みにより中断されたプログラムが再開したときに支障がないように，サービス・ルーチンでは使用したレジスタの退避，回復に気を配らなくてはなりません．

本書の中では，$\overline{\text{IRQ}}$の使用例として，次に示すプリンタ・スプーラ，$\overline{\text{FIRQ}}$の例としては，次章で述べる多重処理のモニタ・プログラムで見ることができます．

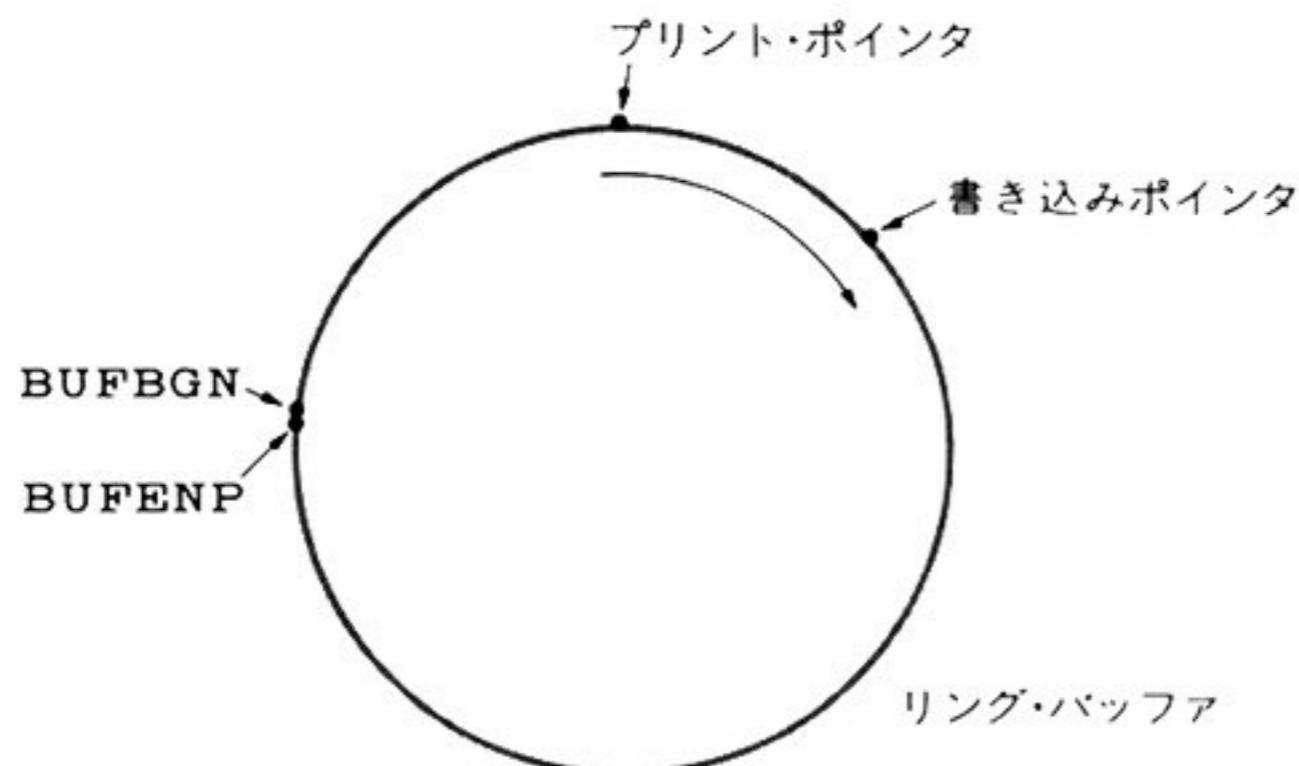
## 6.2 $\overline{\text{IRQ}}$を利用したプリンタ・スプーラ

$\overline{\text{IRQ}}$の応用例としてプリンタ・スプーラを紹介します．スプールとは糸巻のことですが，プリンタに出力すべきデータを糸巻に巻き付けるように溜め込んでおき，プリンタはそれをゆっくりと巻き解すように取り出して印字することからこの名前が付けられたのでしょう．

スタンダードなマイコン・システムの周辺装置としては，プリンタが最も遅い装置です．プログラムの実行がプリンタ待ちのために中断し，いらいらさせられることは，しばしばあります．

計測の結果をプリンタに印字しながら作業を進めるような自動計測システムでは，生産性に影響してしまいます．

この問題を解決するために，最近ではプリンタ・バッファと呼ばれる製品が出回っています．これも概念的にはプリンタ・スプーラと同じもので，マイコンから出力されたデータを大容量のバッファに高速で溜め込んでおき，それをプリンタとハンドシェイクしながら出力するものです．

ここではそれを，$\overline{\text{IRQ}}$を利用してシステムの内部で，ソフト的に実現しようとするものです．

どちらの方法にしても，すべての問題が片付くわけではありません．全体の作業時間に比べて印字に要する時間がはるかに長ければ，いずれはプリンタに歩調を合わせることになってしまいます．しかし，多くの場合は印字が終らなければ次に進めないといったことに問題があるので，たいへん有効なものであることには変わりません．

### ◆ リング・バッファ

スプーラを実現するために，メモリの一部をリング・バッファという形で利用します．この様子を図6.3に示します．メモリの先頭をBUFBGN，最後をBUFENDとラベルを付けることにしておきます．この先頭と最後をプログラムの手段で突き合わせ，リング状のバッファを想定します．

このバッファ内のアドレス・ポインタとして，プリント・ポインタと書き込みポインタ

図6.3 プリンタ・スプーラのリング・バッファの構成

▶BUFBGN はバッファ・メモリの最下位アドレス
▶BUFEND はバッファ・メモリの最上位アドレス

を用意しておきます.

このバッファを利用してプリンタに文字データを出力するわけですが，次に述べるように行われます.

従来，プリンタに1文字を出力していたルーチンは，書き込みポインタの示すアドレスに文字データを書き込み，ポインタを1インクリメントする，ということだけを行います.これだけの作業を行うプログラムを1文字出力ルーチンと見なしてしまうのです.

実際にプリンタにデータを出力するプログラムの実行は，上記のプログラムとは切り離され，割り込みサービス・ルーチンによって行われます.

この割り込みは，プリンタがデータを1文字受け取るたびに起動され，プリント・ポインタが示すアドレスのデータをプリンタに渡します.

プリント・ポインタは書き込みポインタを追い越すことはできないので，プリント・ポインタが書き込みポインタに一致した場合は，バッファは空と判断してなにもせずに終了します.

上記の説明と図からわかるように，このバッファの大きさは，許す限り大きいほうが有利なことはいうまでもありません．その分だけプリンタ待ちとならずに出力できる文字数が多くなります.

プログラム・リストでは，書き込みポインタはBUFPNT，プリント・ポインタはPRINTPとしてラベルが付けられています.

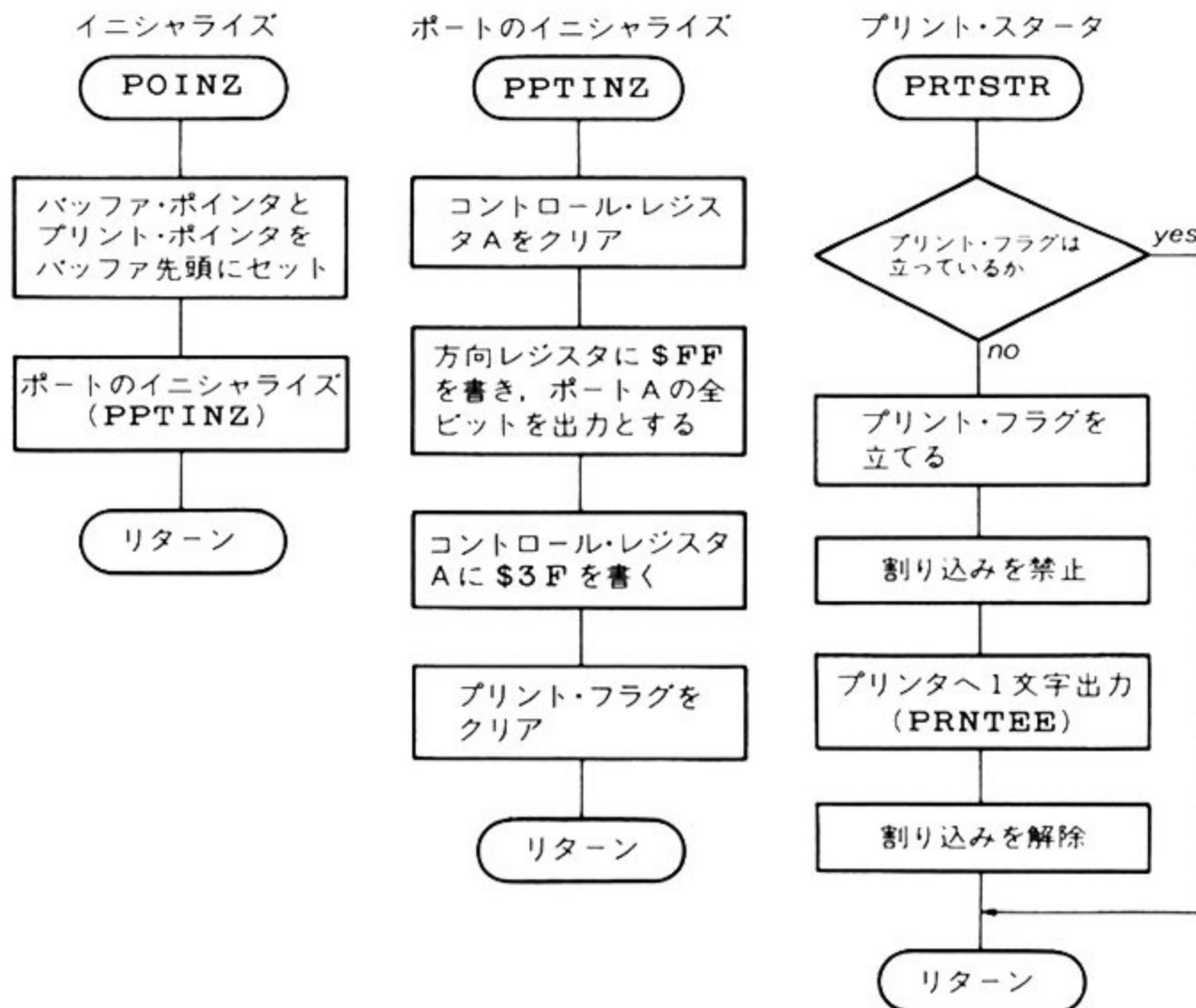
## ◆ プリンタ・スプーラのハードウェア

このための特別なハードウェアが必要なわけではありません．図5.7に示したプリンタ・インターフェースがそのまま使用できます．$\overline{\text{IRQ}}$を使用するので，6821の$\overline{\text{IRQA}}$をMPUの$\overline{\text{IRQ}}$に入力する配線だけを追加してください(図6.2参照)．

## ◆ プリンタ・スプーラのプログラム

このプログラムをリスト6.1に示します．この中で重要な部分は，イニシャライズ，リング・バッファへの書き込み，割り込みサービス・ルーチンの三つに分かれていますが，フロ

リスト6.1　プリンタ・スプーラ

```
00100                          NAM    PRINT-HANDLER
00105                          OPT    M
00110                   **
00120 ECB0                     ORG    $ECB0
00130                   ** PRINT HANDLER WORK AREA **
00140 ECB0 0002         PPT    RMB    2          PORT ADDRES      ┐
00150 ECB2 0002         BUFBGN RMB    2          BUFFER ADDRES    │ ポート・アド
00160 ECB4 0002         BUFEND RMB    2          EXTERNALY INZ    │ レスとバッフ
00170 ECB6 0001         PRNFLG RMB    1          EXT INZ TO 0     │ ァを指定する
00180 ECB7 0002         PRINTP RMB    2          プリント・ポインタ   │ 変数
00190 ECB9 0002         BUFPNT RMB    2          書き込みポインタ     ┘
00200                   **
00210 9000                     ORG    $9000
00220                   ** POINTER INITIALYZE **
00230 9000 BE      ECB2 POINZ  LDX    BUFBGN     バッファとポートのイニシャライズ
00240 9003 BF      ECB7        STX    PRINTP
00250 9006 BF      ECB9        STX    BUFPNT
00260 9009 17      0073        LBSR   PPTINZ
00270 900C 39                  RTS
00280                   **
00290                   ** WRITE DATA INTO BUFFER **   リング・バッファへの書き込み
00300 900D 34      10   WRBUF  PSHS   X
00310 900F BE      ECB9        LDX    BUFPNT
00320 9012 A7      80          STA    ,X+        リング・バッファにデータをライト
00330 9014 BC      ECB4        CMPX   BUFEND  ┐
00340 9017 23      03          BLS    WRBUF1  │ ポインタがBUFENDならBUFBGN
00350 9019 BE      ECB2        LDX    BUFBGN  ┘ に書き換える
00360 901C BF      ECB9 WRBUF1 STX    BUFPNT
00370 901F 81      0D          CMPA   #$0D    ┐
00380 9021 26      02          BNE    WRBUF2  │ 書き込んだデータがCRコードならプリント
00390 9023 8D      44          BSR    PRTSTR  ┘ をスタート
00400 9025 BC      ECB7 WRBUF2 CMPX   PRINTP     書き込みポインタとプリント・ポインタを
00410 9028 26      06          BNE    WRBUF3     比較．一致していなければ終了
00420 902A 8D      4E          BSR    PAUSE   ┐
00430 902C 8D      3B          BSR    PRTSTR  │ 一致の場合はここで待つ
00440 902E 20      F5          BRA    WRBUF2  ┘
00450 9030 35      90   WRBUF3 PULS   X,PC
00460                   **
00470                   **
```

### リスト6.1 プリンタ・スプーラ (つづき)

```
00480                      ** PRINT EE **           プリンタへ1文字出力
00490 9032 34      32      PRNTEE PSHS  A,X,Y
00500 9034 BE      ECB7           LDX   PRINTP
00510 9037 A6      80             LDA   ,X+         プリント・ポインタの示すデータをAにロード
00520 9039 BC      ECB4           CMPX  BUFEND  ┐
00530 903C 23      03             BLS   PRNT2   ├ ポインタがBUFENDならBUFBGN
00540 903E BE      ECB2           LDX   BUFBGN  ┘ に書き換える
00550 9041 10BE    ECB0    PRNT2  LDY   PPT
00560 9045 A7      A4             STA   ,Y
00570 9047 6D      A4             TST   ,Y                DUMMY READ
00580 9049 BF      ECB7           STX   PRINTP
00590 904C 86      35             LDA   #$35    ┐
00600 904E A7      21             STA   1,Y     ├ ストローブ
00610 9050 86      3D             LDA   #$3D    │
00620 9052 A7      21             STA   1,Y     ┘
00660 9054 35      B2             PULS  A,X,Y,PC
00670                      **
00680                      ** PRINT IRQ HANDLER **  IRQの割り込みサービス・ルーチン
00690 9056 BE      ECB7    INTPRN LDX   PRINTP
00700 9059 BC      ECB9           CMPX  BUFPNT    プリント・ポインタと書き込みポインタを比較.
00710 905C 27      03             BEQ   ENDPRN    一致していれば終了
00720 905E 8D      D2             BSR   PRNTEE
00730 9060 3B                     RTI
00740 9061 7F      ECB6    ENDPRN CLR   PRNFLG
00745 9064 6D9F    ECB0           TST   [PPT]
00750 9068 3B                     RTI
00760                      **
00770                      **
00780                      ** PRINTER STARTER **      プリントのスタート
00790 9069 7D      ECB6    PRTSTR TST   PRNFLG
00800 906C 26      0B             BNE   PRTST1    プリントが起動していればリターン
00810 906E 73      ECB6           COM   PRNFLG
00820 9071 34      01             PSHS  CC
00830 9073 1A      50             ORCC  #$50
00840 9075 8D      BB             BSR   PRNTEE    1文字出力を実行する
00850 9077 35      01             PULS  CC
00860 9079 39              PRTST1 RTS
00870                      **
00880                      ** PAUSE **
00890 907A 39              PAUSE  RTS
00900 907B 12                     NOP
00910 907C 12                     NOP
00920 907D 12                     NOP
00930 907E 12                     NOP
00940                      **
00950                      * PRINT PORT INITIALYZE *   ポートのイニシャライズ
00960 907F 34      12      PPTINZ PSHS  A,X
00970 9081 BE      ECB0           LDX   PPT
00980 9084 6F      01             CLR   1,X       PIAのCRAをクリア
00990 9086 86      FF             LDA   #$FF
01000 9088 A7      84             STA   ,X        PAを全ビット出力
01010 908A 86      3F             LDA   #$3F
01020 908C A7      01             STA   1,X       コントロール・コードのライト
01025 908E 7F      ECB6           CLR   PRNFLG    プリント・フラグをクリア
01030 9091 35      92             PULS  A,X,PC
01040                      **
01050                      **
01060                             END
```

図6.4　プリンタ・スプーラのフローチャート(1)

ーチャートを使用して順に説明します．

このプログラムの本体部分は，アドレス・インディペンデント(どのアドレスに配置してもアセンブルしなおすことなく実行できる)であるほか，リング・バッファのアドレス，大きさ，さらにポート(6821)のアドレスも自由に変更でき，これらの値を変数に書き込んでからこのプログラムを使用するように組まれています．この変数のアドレスだけは固定アドレスになっています．必要があれば別の値にしてアセンブルしなおしてください．

図6.4は，イニシャライズのフローチャートです．プログラム・リストでは220行の部分です．プリンタ・スプーラの使用を開始する前に，1回だけPOINZを実行しなくてはなりません．

ここでは，リング・バッファの初期化とポートのイニシャライズも行っています．

ポートのイニシャライズ(PPTINZ)の最後でクリアされているプリント・フラグとは，プリンタへのデータ転送が続行中であることを示す変数であり，プログラム・リストでのラベル名はPRNFLGです．

図6.5
プリンタ・スプーラのフローチャート(2)
(リング・バッファへの書き込み)

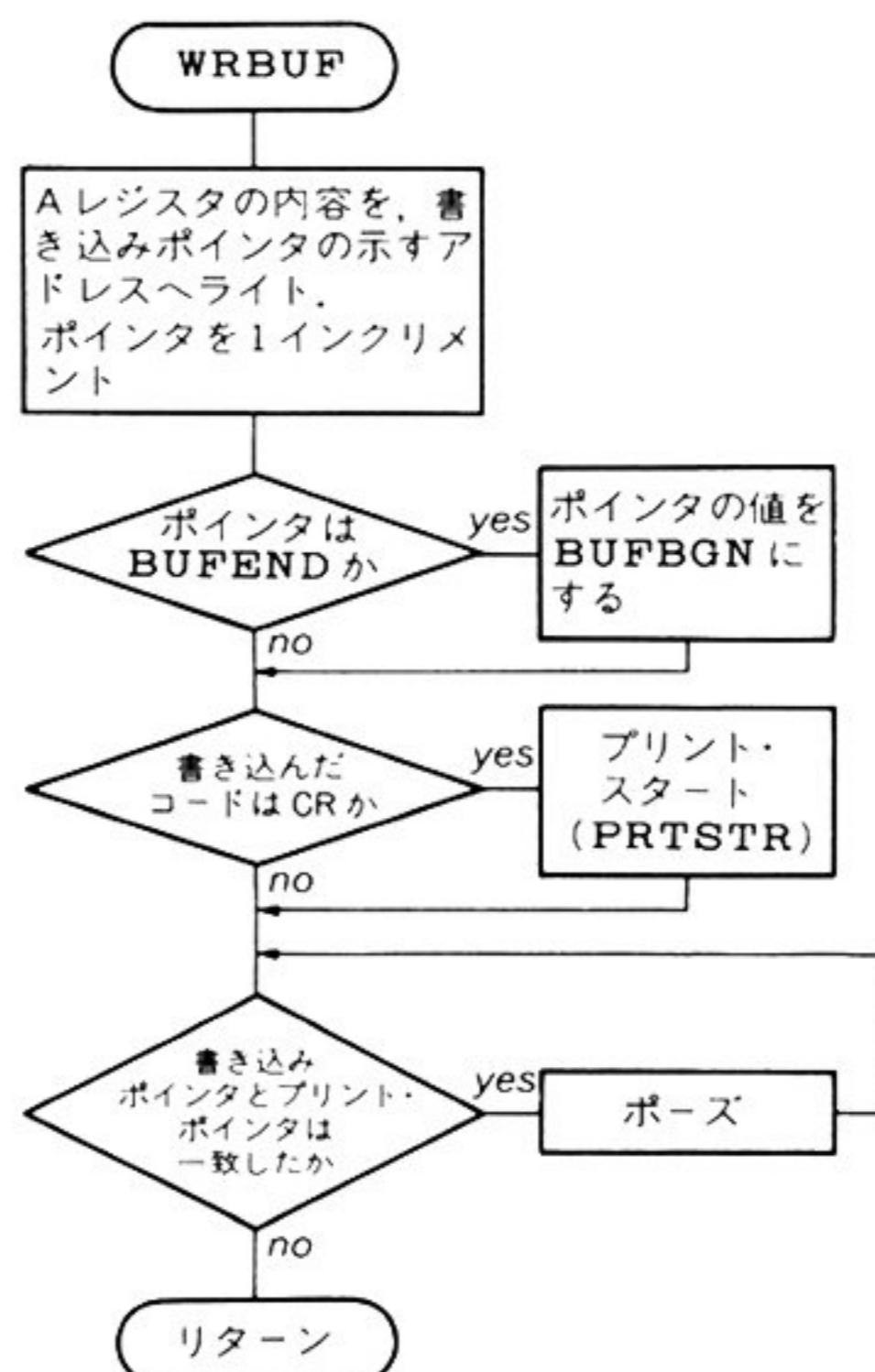
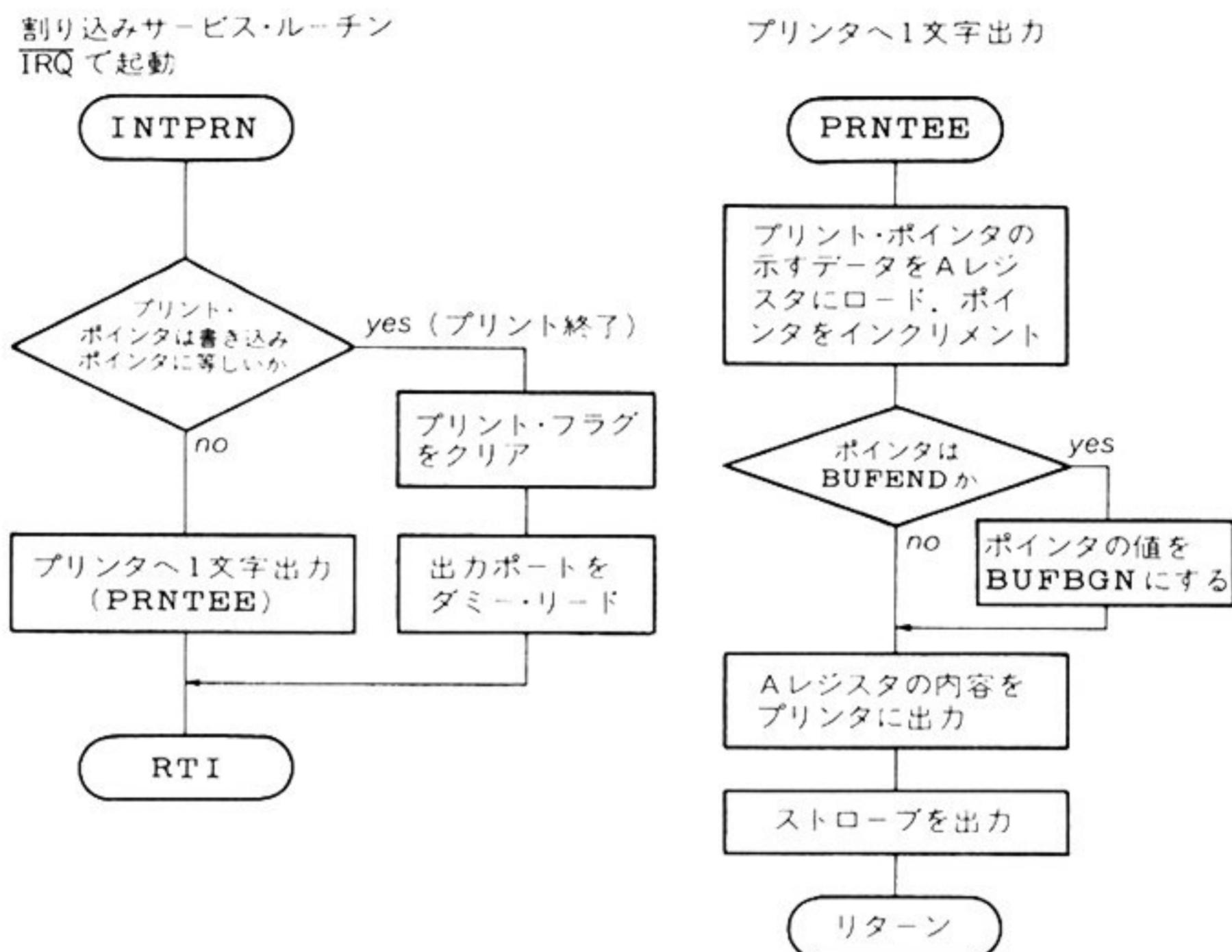



図6.5は，リング・バッファへの書き込みを示すフローチャートです．リストでは290行からです．

プリンタへの出力を行うユーザ・プログラムでは，1文字出力ルーチンの代わりにこのサブルーチン(WRBUF)を使います．

このフローチャートで，バッファの先頭と最後が連結され，リング状になっていることがわかると思います．つまり，ポインタがBUFENDの値になったら，先頭であるBUFBGNの値に強制的に書き換えます．

書き込んだデータがキャリッジ・リターン(CR)の場合には，プリント・スタート・ルーチン(PRTSTR)を実行します．バッファに書いただけでは印字を開始しないので，どこかでプリンタへのデータ転送を開始させる必要があります．このきっかけをCRコードとしました．

転送の開始は，1文字をプリンタに転送することで行えます．

プリンタへのデータ転送がいったん始まると，プリンタからのアクノレッジ信号で$\overline{\text{IRQ}}$が発生し，割り込みサービス・ルーチンにより次のデータが転送されます．この転送によ

図6.6　プリンタ・スプーラのフローチャート(3)

り，再び$\overline{\text{IRQ}}$が発生します．このようにしてバッファへの書き込みとは独立して，プリンタへの転送が繰り返されることになり，バッファが空になるまで続けます．

WRBUFの最後の部分では，書き込みポインタとプリント・ポインタの一致を調べています．これは，書き込みポインタがバッファを一巡して後からプリント・ポインタに追いついてしまったかをテストします．

追いついてしまっていれば，この時ばかりはバッファへの書き込みが待たされます．プリンタへの転送が進行してプリント・ポインタが進むのを待つほかありません．

この待ちループにあるポーズとは，後で述べるマルチ・ジョブのシステムとした場合の用意であり，MPUがこのループのためにむだな時間を費やさないように，一時的にMPUをほかのプログラムに明け渡すことです．紹介したリストでは，この用意がないことを前提として，ポーズの部分はNOP(ノー・オペレーション：何もしない)としてあります．

図6.6は，割り込みサービス・ルーチンの内容です．これがプリンタのアクノレッジによって生じた$\overline{\text{IRQ}}$により起動される部分です．

まず，プリント・ポインタと書き込みポインタが一致しているかを調べます．プリント・

ポインタが書き込みポインタに追いついて一致したとは，バッファが空になったということです．この場合は，プリント・フラグのクリアと 6821 の $\overline{\text{IRQA}}$ をクリアするためのダミー・リードを行って終了します．

まだ一致していなければ，プリンタへ1文字を転送します．1文字出力(PRNTEE)も見てみましょう．

ここでも，バッファをリング状とする手続きがとられています．プリント・ポインタがBUFEND に達したら，ポインタの値を強制的にバッファの先頭 BUFBGN にします．

ストローブを出力した後に，アクノレッジ待ちのループがないことに注意してください．

### ◆ プリンタ・スプーラの使い方

これまでの説明で，プリンタ・スプーラの概要は理解できたと思います．ここでは，具体的な使用方法について説明します．

まず，プリンタへの出力を伴っていたプログラムですが，プリンタへの出力ルーチンとした部分を**リスト6.1**の WRBUF に代えます．

つまり，**リスト5.7**で示した PRNTEE を使用して，JSR　PRNTEE としていたのであれば，これを JSR　WRBUF に書き換えます．

次に，$\overline{\text{IRQ}}$ のベクタ・アドレスに，割り込みサービス・ルーチンである INTPRN(リスト 690 行)のアドレスを書き込みます．この場合，直接のベクタ・アドレスである$FFF8 に書いてもよいのですが，普通は，$\overline{\text{IRQ}}$ によってモニタ・プログラムの一部が起動して，モニタが定めた特定の RAM 域に書かれている内容をベクタ・アドレスとしてジャンプする，といった方法が取られます．使用するモニタの説明などを参照してください．ASSIST09(モトローラ社)の場合は，モニタのベクタ・スワップ・サービスを利用して INTPRN のアドレス値をベクタに設定します．

最後に，ポートとバッファのアドレスを指定します．6821 が割り付けられた PA のアドレスを PPT(リスト 140 行)のアドレスに書き込み，バッファとして使用するメモリ・エリアの先頭アドレスを BUFBGN(リスト 150 行)，最終アドレス+1 を BUFEND(リスト 160 行)の示すアドレスに書き込みます．

バッファ・アドレスやサイズに制限はなく，バッファのサイズは大きいほうがプリンタ待ちとなる機会が少なくなることはいうまでもありません．

以上のことがプログラムとして用意できれば準備は完了です．これでユーザ・プログラムを実行すれば，プリンタ待ちとなる機会はずっと減少してシステムの効率がぐんと高く

なります．

このプリンタ出力は$\overline{\text{IRQ}}$を使用するわけですから，メイン・プログラムではプリンタの使用を開始する以前に，CCRのIRQマスク・ビットをクリアしてからプリンタ・スプーラを実行させることも忘れないでください．

## 6.3 SWIによるシステム・コールの方法

これまでに説明した割り込みの例は，MPUの外部から，ハード的に割り込み要求が発生して割り込みサービス・ルーチンを起動させるものでした．

ここでは，プログラム命令によって割り込みルーチンのサービスを受ける，ソフトウェア・インタラプト命令の応用例を紹介します．

このような命令として6809では，SWI，SWI2，SWI3の三つが用意されています．この命令の詳細説明については，第4章のニーモニック説明，SWIの項を参照してください．

ミニコンを経験された方は，トラップ命令の単純な例と考えるとわかりやすいと思います．68000の場合も，TRAP命令にすべて含まれており，システム・コールもTRAP命令で行います．

**● システム・コールについて**

システム・コールとは，モニタ・プログラムやOS(オペレーティング・システム)のもつ機能の一部をユーザ・プログラムでこれをコールして利用することをいいます．

このコールの方法ですが，サブルーチン・ジャンプ(JSRまたはBSR)によるのが考え方としては最も簡単ですが，この場合には相手のアドレスを知らなければならず，互いに位置独立という点でも問題が生じてきます．この方法は小規模なシステム向きというべきでしょう．

これに代えて，TRAP命令や6809の場合ではSWI命令を使用した方法では，相手のサービス・ルーチンのアドレスを知る必要はまったくなく，互いにどのアドレスに配置されているとしても，何の変更もなくまったく同様にサービスを受けることができます．

68000では，このTRAP命令が強力なため，階層化された大型のシステム・ソフトで階層間でのコールに数種類のTRAP命令を使用して，ブロックの独立性を高め，プログラムの開発効率も向上するように考慮されています．

6800ではSWIの一つだけだったものが，6809ではSWI2とSWI3が追加されたのも，同様なことを考えてのことと思います．

ここで紹介するシステム・コールの例は，システムの最も基本である，ターミナルの入出力を行うSWIサービス・プログラムとその使用方法です．

このプログラムがサポートするのは，ターミナル入出力の九つの基本的な機能です．それぞれの機能はコード番号で区別され，コード番号と機能を以下に示します．

0　ターミナルから1文字をAレジスタに取り込む
1　Aレジスタの文字をターミナルに出力する
2　キャリッジ・リターンとライン・フィードおよびXレジスタで示す文字列をターミナルに出力する
3　Xレジスタで示す文字列をターミナルに出力する
4　Xレジスタで示す1バイトを16進2桁で表示する
5　Xレジスタで示す1ワードを16進4桁で表示する
6　キャリッジ・リターンとライン・フィードをターミナルに出力する
7　ブランクを出力する
8　モニタ・プログラムにもどる

この機能を利用する方法は，SWI命令と，それに続く1バイトのコード番号で指定します．例えば，1バイトの16進数出力であれば以下のようになります．

```
SWI
FCB  4
```

これはJSR　OUT2H(OUT2Hは16進2桁を出力するサブルーチン)と同じ結果になります．

### ◆ サービス・プログラム

それでは，以上のような機能を実現するプログラムとはどのようなものかを説明します．**リスト6.2**に，このプログラム例を示します．

このプログラムでは，ターミナルの入出力として紹介した**リスト5.1**から**リスト5.5**までをサブルーチンとして使用しますので，一緒にアセンブルする必要があります．

1820行のSWIVECが，SWIサービス・ルーチンの開始になります．つまり，ここのアドレス値をSWIのベクタ・アドレスに設定しておきます．

SWIVECはSWI命令の次に置かれたコード番号を読み出し，テーブル(1940行から)を使用して計算した目的とするルーチンのアドレスにジャンプします．

## リスト6.2　SWIハンドラ

```
01410                        **
01420                        **
01430                        **
01440                        **
01450                        * SWI HANDLER *
01460 00B7 17       FF52 INCHP   LBSR   INCH
01470 00BA A7       61           STA    1,S       1文字入力
01480 00BC 3B                    RTI
01490                        **
01500 00BD A6       61   OUTCHP  LDA    1,S       1文字出力
01510 00BF 17       FF7A         LBSR   OUTCH
01520 00C2 3B                    RTI
01530                        **
01540 00C3 AE       64   PDATA1  LDX    4,S       文字列の出力．キャリッジ・リターンを出力
01550 00C5 8D       60           BSR    CRLF      してから文字列を出力
01560 00C7 17       FF81         LBSR   BUFOUT
01570 00CA 3B                    RTI
01580                        **
01590 00CB AE       64   PDATA   LDX    4,S       文字列の出力
01600 00CD 17       FF7B         LBSR   BUFOUT
01610 00D0 3B                    RTI
01620                        **
01630 00D1 AE       64   OUT2HS  LDX    4,S       Xの示す1バイトを16進2桁で出力
01640 00D3 A6       84           LDA    ,X
01650 00D5 8D       32           BSR    OUT2H
01660 00D7 3B                    RTI
01670                        **
01680 00D8 AE       64   OUT4HS  LDX    4,S       Xの示す2バイトを16進4桁で出力
01690 00DA EC       84           LDD    ,X
01700 00DC 8D       43           BSR    OUT4H
01710 00DE 3B                    RTI
01720                        **
01730 00DF 8D       46   PCRLF   BSR    CRLF      キャリッジ・リターンを出力
01740 00E1 3B                    RTI
01750                        **
01760 00E2 8D       4E   SPACE   BSR    OUTSP     スペース・コードを出力
01770 00E4 3B                    RTI
01780                        **
01790 00E5 7E       F800 MONITR  JMP    $F800
01800                        **
01810                        **
01820 00E8 A6       E4   SWIVEC  LDA    ,S      } CCRを回復
01830 00EA 1F       8A           TFR    A,CC    }
01840 00EC AE       6A           LDX    10,S      PCをXにリード
01850 00EE A6       01           LDA    1,X       コード番号をAにリード
01860 00F0 30       02           LEAX   2,X
01870 00F2 AF       6A           STX    10,S      Xの値を退避したPCのアドレスにストア
01880 00F4 308D     0004         LEAX   TBL,PCR
01890 00F8 E6       86           LDB    A,X
01900 00FA 6E       85           JMP    B,X
01910                        **
01920                        **
```

リスト6.2 SWIハンドラ (つづき)

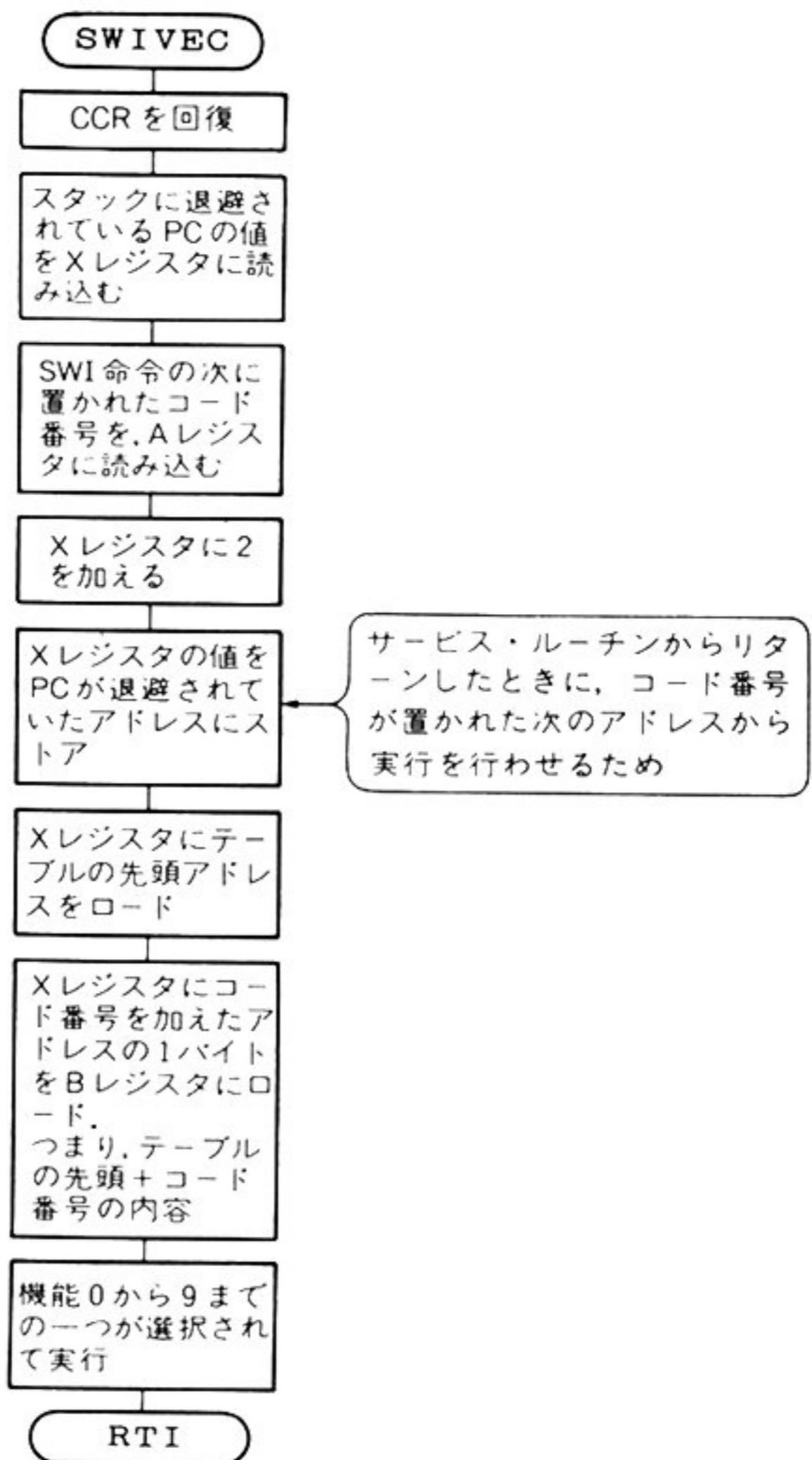
```
01930                         * HANDLER TABLE **
01940 00FC BB                 TBL     FCB   INCHP-TBL    0 ┐
01950 00FD C1                         FCB   OUTCHP-TBL   1 │
01960 00FE C7                         FCB   PDATA1-TBL   2 │
01970 00FF CF                         FCB   PDATA-TBL    3 │
01980 0100 D5                         FCB   OUT2HS-TBL   4 ├コード番号
01990 0101 DC                         FCB   OUT4HS-TBL   5 │
02000 0102 E3                         FCB   PCRLF-TBL    6 │
02010 0103 E6                         FCB   SPACE-TBL    7 │
02020 0104 E9                         FCB   MONITR-TBL   8 ┘
02030 0105 00                         FCB   0
02040 0106 00                         FCB   0
02050 0107 00                         FCB   0
02060 0108 00                         FCB   0
02070                         **
02080                         **
02090                         **
02100                         * OUT-PUT 2 HEX CHR. *     16進2桁出力
02110 0109 34      02         OUT2H   PSHS  A
02120 010B 44                         LSRA
02130 010C 44                         LSRA
02140 010D 44                         LSRA
02150 010E 44                         LSRA
02160 010F 8D      04                 BSR   OUTFIG
02170 0111 35      02                 PULS  A
02180 0113 84      0F                 ANDA  #$0F
02190 0115 8B      30         OUTFIG  ADDA  #'0
02200 0117 81      3A                 CMPA  #$3A
02210 0119 25      02                 BCS   OUTFG2
02220 011B 8B      07                 ADDA  #7
02230 011D 17      FF1C       OUTFG2  LBSR  OUTCH
02240 0120 39                         RTS
02250                         **
02260                         **
02270                         * OUT-PUT 4 HEX CHR. *     16進4桁出力
02280 0121 8D      E6         OUT4H   BSR   OUT2H
02290 0123 1F      98                 TFR   B,A
02300 0125 20      E2                 BRA   OUT2H
02310                         **
02320                         **
02330 0127 86      0D         CRLF    LDA   #$0D
02340 0129 17      FF10               LBSR  OUTCH
02350 012C 86      0A                 LDA   #$0A
02360 012E 17      FF0B               LBSR  OUTCH
02370 0131 39                         RTS
02380                         **
02390                         **
02400 0132 34      02         OUTSP   PSHS  A
02410 0134 86      20                 LDA   #$20
02420 0136 17      FF03               LBSR  OUTCH
02430 0139 35      82                 PULS  A,PC
02440                         **
02450                         **
02460                         **
02470                         **
```

リスト6.2 SWIハンドラ（つづき）

```
02480                        * MULTI TASK SAMPLE PROGRAM *
02490                        *
02500         4009           DELAY   EQU    $4009
02510                        **
02520 013B 86      41        TASK1   LDA    #'A       タスク1
02530 013D 17      FEFC      TASK11  LBSR   OUTCH     "A"を連続してターミナルに出力
02540 0140 20      FB                BRA    TASK11
02550                        **
02560                        **
02570 0142 8E      E010      TASK2   LDX    #ACIAC    タスク2
02580 0145 CC      0005      KEY     LDD    #5        キー入力した文字コードを
02590 0148 36      06                PSHU   D         そのままプリンタに出力
02600 014A BD      4009              JSR    DELAY
02610 014D E6      84                LDB    ,X
02620 014F 57                        ASRB
02630 0150 24      F3                BCC    KEY
02640 0152 A6      01                LDA    1,X
02650 0154 84      7F                ANDA   #$7F
02660 0156 27      ED                BEQ    KEY
02670 0158 17      FF06              LBSR   PRNTEE
02680 015B 20      E8                BRA    KEY
02690                        **
02700                        **
02710                                END
```

テーブルには，それぞれのサービス・ルーチンとテーブル先頭アドレスの差，すなわち，テーブル先頭アドレスからのオフセット値が，コード番号の順に並んでいます．

このプログラムの説明を，図6.7にフローチャートで示しておきました．

それぞれのサービス・ルーチンは，リストでその内容は容易にわかると思いますが，簡単に補足しておきます．

**INCHP（1文字入力，1460行）**

リスト5.2の1文字入力ルーチンを使っています．次の行の命令では，Aレジスタに得られた文字データを，Aレジスタが退避されているスタックのアドレスに書き込みます．これで，RTIでサービス・ルーチンからリターンしたときには，Aレジスタに入力されたデータが格納されていることになります．

つまり，SWI命令の割り込みサービス・ルーチンからRTIでリターンした場合には，すべてのレジスタは，スタックに退避されているものが再びもどされるわけですから，レジスタに新しいデータをもたせてリターンさせたい場合は，退避先のアドレスに書き込む必要があるのです．

図6.7
SWIサービス・ルーチンのフローチャート

**OUTCHP(1文字出力，1500行)**

リスト5.4の1文字出力ルーチンを使っています．レジスタのスタックへの退避については,INCHPの場合とは逆に,退避されているAレジスタの内容をスタックから拾ってこなくてはなりません．これがLDA　1，Sです．

**PDATA1(改行して文字列の出力，1540行)**

リスト5.5のバッファ出力ルーチンを使っています．ここでは，文字列のポインタとしてXレジスタを必要としますので，退避されているXレジスタの内容をスタックから読み出

してから，CRLF と BUFOUT を実行します．文字列の終了は EOT($04) です．

CRLF はキャリッジ・リターンとライン・フィード・コードを出力するサブルーチンです．2330行にそのプログラムがあります．

### PDATA(文字列の出力，1590行)

Xレジスタで示す文字列の出力だけを行います．PDATA1 から CRLF を除いたものです．

### OUT2HS(16進2桁の出力，1630行)

Xレジスタで示す1バイトのデータを，16進2桁の数字として出力します．

ここでも退避されたXレジスタをスタックから読み出して，ポインタとして使用します．

OUT2H は，Aレジスタの内容を16進2桁の ASCII コードに変換してターミナルに出力します．リストの2110行を参照してください．

### OUT4HS(16進4桁の出力，1680行)

Xレジスタで示す2バイトのデータを16進4桁で出力します．

OUT4H はDレジスタの内容を16進4桁の ASCII コードに変換して出力するサブルーチンであり，OUT2H を利用して作ってあります．リストの2280行を参照してください．

### PCRLF(改行，1730行)

キャリッジ・リターンのライン・フィード・コードを出力します．CRLF は2330行を参照してください．

### SPACE(空白，1760行)

スペース・コード($20)を出力します．OUTSP は2400行を参照してください．

### MONITR(モニタに入る，1790行)

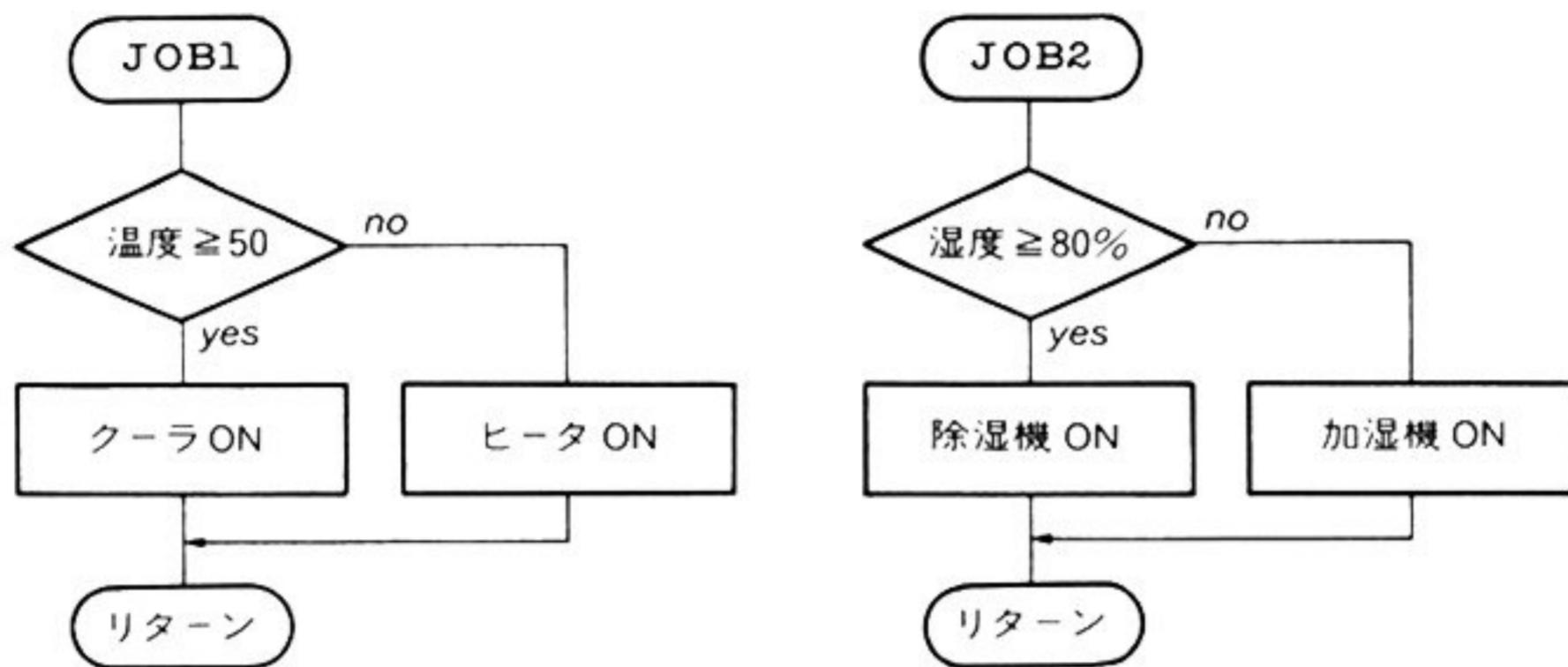
モニタ・プログラムに実行をもどします．ジャンプ先の$F800は使用するモニタ・プログラムの開始番地，または，ウォーム・スタートがあればそのアドレスに書き換えてください．

# 第7章
# 多重処理とマルチ・タスク・モニタ

マイコンのハードウェアとアセンブラ・プログラミングをようやく覚えた方にとっては，多重処理やマルチ・タスクと聞くと，特別な高度の技術を要する難解なものだと思われるかも知れません．

マイコンの応用技術としては，レベルの高いものには違いありませんが，かといって，初心者にとってまったく縁のないものでも，使いこなせないものでもないと思います．

この章では，多重処理についての基礎的な概念の説明と，簡単なマルチ・タスク・モニタを紹介します．

マルチ・タスク・モニタというと，規模も大きくそのマニュアルを読むだけでもひと苦労するものが多いのですが，ここで紹介するものは，プログラム・サイズも700バイト程度であり，大変小さなものです．そのため，小規模な6809システムに容易に組み込むことができ，わずかの約束事を理解すれば，多重処理のプログラミングが可能になります．

ミニ・サイズのマルチ・タスク・モニタであるので，最低限の機能しかありませんが，これでも自動計測や制御装置に組み込まれて十分に実績のあるものです．

## 7.1　多重処理とは

多重処理の経験がなくても，リアルタイムOS，リアルタイム・モニタ，マルチ・タスク，マルチ・ジョブなどといった言葉は聞いたことがあると思います．これらの核を成すのが多重処理なのです．

多重処理とは，二つ以上の仕事を見掛け上は同時に実行処理していくことをいいます．この仕事が，タスクまたはジョブと呼ばれます．

図7.1　恒温槽制御のフローチャート

この二つの言葉は，厳密には区別される場合もあるようですが，ほとんど同じ意味であり，プログラムが実行して処理する仕事のことを呼びます．ここではタスクということにします．

見掛け上は，といったのは，瞬間の時間で見れば，プロセッサが実行するタスクは一つでしかありませんが，二つのタスクをごく短い時間で区切って交互に実行したとしたら，この二つのタスクは同時に並行して処理されているように見えるはずです．

これを時分割処理と呼びますが，これが多重処理を実現させる最初の概念になります．

ミニコン以上のシステムでは，マルチ・ユーザ・システムとなっているのが普通ですが，これも多重処理ということでは同様であり，これから話題にする多重処理よりも規模がずっと大きいだけです．

このようなシステムでは，高速のプロセッサや記憶装置などの演算処理に必要な本体(メイン・フレームと呼ぶ)は1台で，複数の利用者が複数の端末から同時に使用します．しかもこの場合に，どの利用者から見ても，メイン・フレームは自分だけのために働いているように見せ掛けることができます．

それでは，多重処理の最も簡単な例から見てみましょう．

## 7.2　恒温槽をコントロールする例を考える

恒温槽のようなものを考え，温度を50度，湿度を80％一定に保つプログラムを考えます．

図7.1のフローチャートを見てください．JOB1は温度を一定にするためのサブルーチ

図7.2 待ち要素のあるプログラム

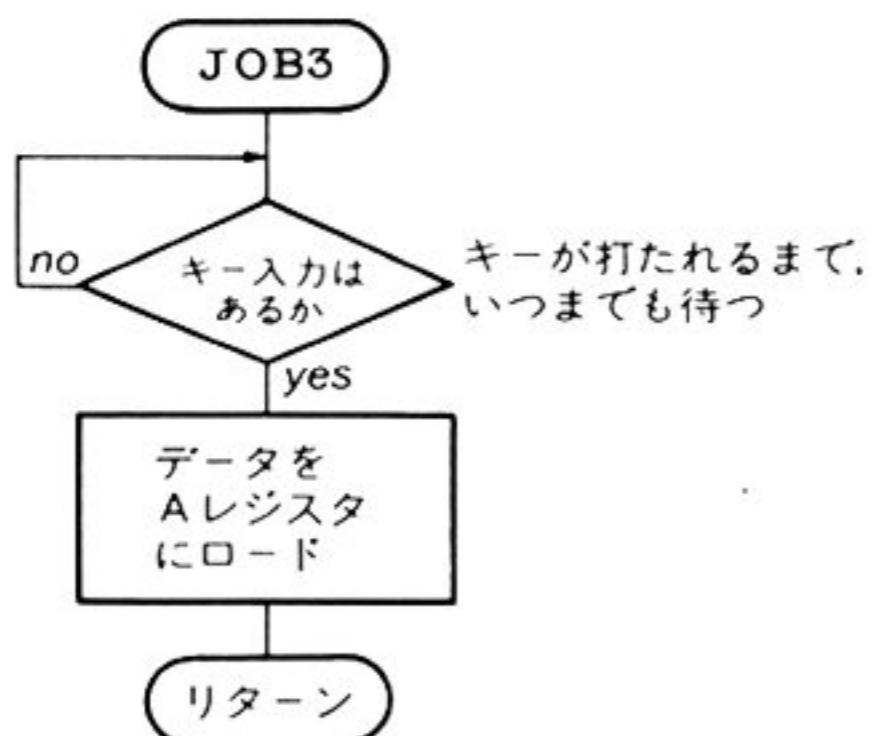
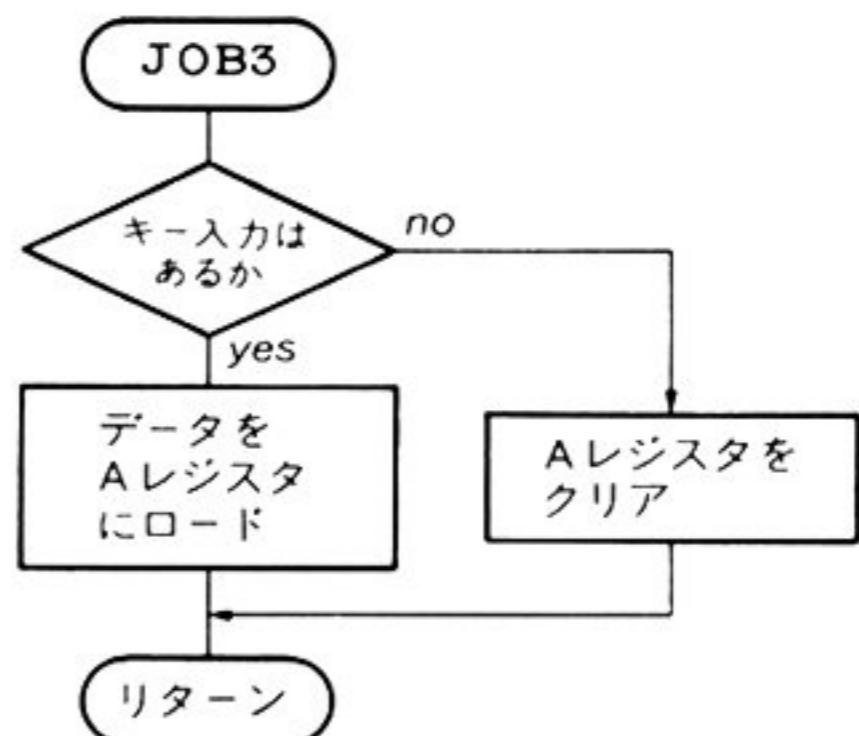



図7.3 ループを解いたフローチャート

ンであり，クーラとヒータを制御しています．JOB2 は，除湿機と加湿機を操作して湿度制御を行うサブルーチンです．

この二つサブルーチンを使用した，次のプログラムを実行したとしましょう．

```
CNTRL   JSR   JOB1
        JSR   JOB2
        BRA   CNTRL
```

JOB1 と JOB2 の実行時間はせいぜい十数 $\mu$s 程度でしょう．これが交互に実行されていれば，私たちの目には，温度制御と湿度制御の二つの作業が同時に行われているように見えます．これはすでに多重処理の形を取っていることになります．

## ◈ 待ち要素に対する処置

今度は，温度と湿度を同時に制御しながら，キーボードの“S”が打たれたら作業を中止する，“S”以外ならばその文字を出力するというプログラムを考えてみましょう．

この場合，キー入力のルーチンとして，第5章で紹介した INCH を使ったとしたらどうでしょう．INCH のフローチャートを少し簡素化して図7.2 に示します．このプログラムはキー入力があるまで，いつまでもループを回り続けます．

これでは先のように，JOB1, JOB2, JOB3 を短い時間で交互に実行することが不可能になります．

そこでループを解いたフローチャートを考えます．それを図7.3 に示します．今度は，キー入力がなかった場合はキー入力待ちのループを実行するのでなく，Aレジスタをクリア

してフラグとして残し，そのままリターンします．

図7.3のJOB3を使ったプログラムは，次のようになります．

```
CNTRL   JSR     JOB1
        JSR     JOB2
        JSR     JOB3
        TSTA
        BEQ     CNTRL
        CMPA    #'S
        BEQ     STOP
        JSR     OUTCH
        BRA     CNTRL
STOP    RTS
```

これならば，問題とするほどの待ち時間が発生することなく，温度制御，湿度制御，キー入力の処理の三つの作業を並行して行うことができます．

このように，多重処理のプログラムでは，待ち要素に対する処置が重要な問題であることがわかると思います．とくにマイコン応用の例では，待ち要素を含む場合がたいへん多く，これをどう処置するかが並行処理を行う決め手でもあり，フラグを上手に使った方法がいろいろと考えられていますが，規模が大きくなるとそれも困難になってきます．

そこで威力を発揮してくれるのが，後で紹介するマルチ・タスク・モニタなのです．

## 7.3　割り込みによる多重処理

マイコンで多重処理といえば，すぐに割り込み処理を連想するのが普通です．割り込み処理と多重処理とは密接な関係があるのです．

恒温槽の例は，プログラムの方法を工夫することで多重処理を行いましたが，このような例はごく普通のプログラムの中でも頻繁に見られることで，改めて多重処理というほどのものでもありません．

一般に多重処理と呼ぶ場合は，外見的に独立した作業を並行処理するだけでなく，プログラムそのものも互いに独立性が高く，それを並行して実行処理することを想定しています．このようなことは，割り込みを利用することで容易に実現できます．

図7.4 プリンタ・スプーラが実行するタイミング

割り込みを使った多重処理の例は，本書の中でもすでに紹介してあり，プリンタ・スプーラがその例です．プリンタ・スプーラが実行する様子を，もう一度ここで考えてみましょう．

バッファへの文字列の書き込みはすでに終了し，プログラムは次の作業を実行中ですが，バッファに溜った文字列をプリンタの速度に合わせて1字ずつプリンタに転送するタイミングを考えます．

この様子を図7.4に示しました．プリンタのアクノレッジによってIRQが発生し，実行中のプログラムは一時中断してプリンタ・スプーラの割り込みサービス・ルーチンが起動し，1文字を転送します．転送が終われば元のプログラムの実行を再開します．

以上の動作を繰り返しているわけですが，この時点では，実行中のプログラムと割り込みサービス・ルーチンによる転送作業とでは，外見的な作業内容も，互いのプログラムも独立したものになっています．

すなわちこれは，恒温槽の例よりも少しレベルの高い多重処理ということができます．

プリンタ・スプーラの例では，割り込みを要求する要素は一つだけですが，複数の場合のことを考えておきます．

6809の割り込み入力は$\overline{\mathrm{IRQ}}$, $\overline{\mathrm{FIRQ}}$, $\overline{\mathrm{NMI}}$の3本があります．これを使い分けることもできますが，これらの割り込みはそれぞれに性格が異なるので，ここでは$\overline{\mathrm{IRQ}}$だけを考えることにします．

● **ポーリングによって割り込み源を探す**

第3章で紹介したCPUボードの例では，$\overline{\mathrm{IRQ}}$割り込みの拡張回路が組み込まれています．これを利用すれば，八つまでの割り込み要求に対して，それぞれが別のベクタ・アドレスをもつことができるので，独立した割り込みサービス・ルーチンが用意できます．

しかし，このような拡張回路はない場合のほうが多いので，その場合には割り込みの要求先はどこなのかを，サービス・ルーチンで探らなくてはなりません．これを行うのが，

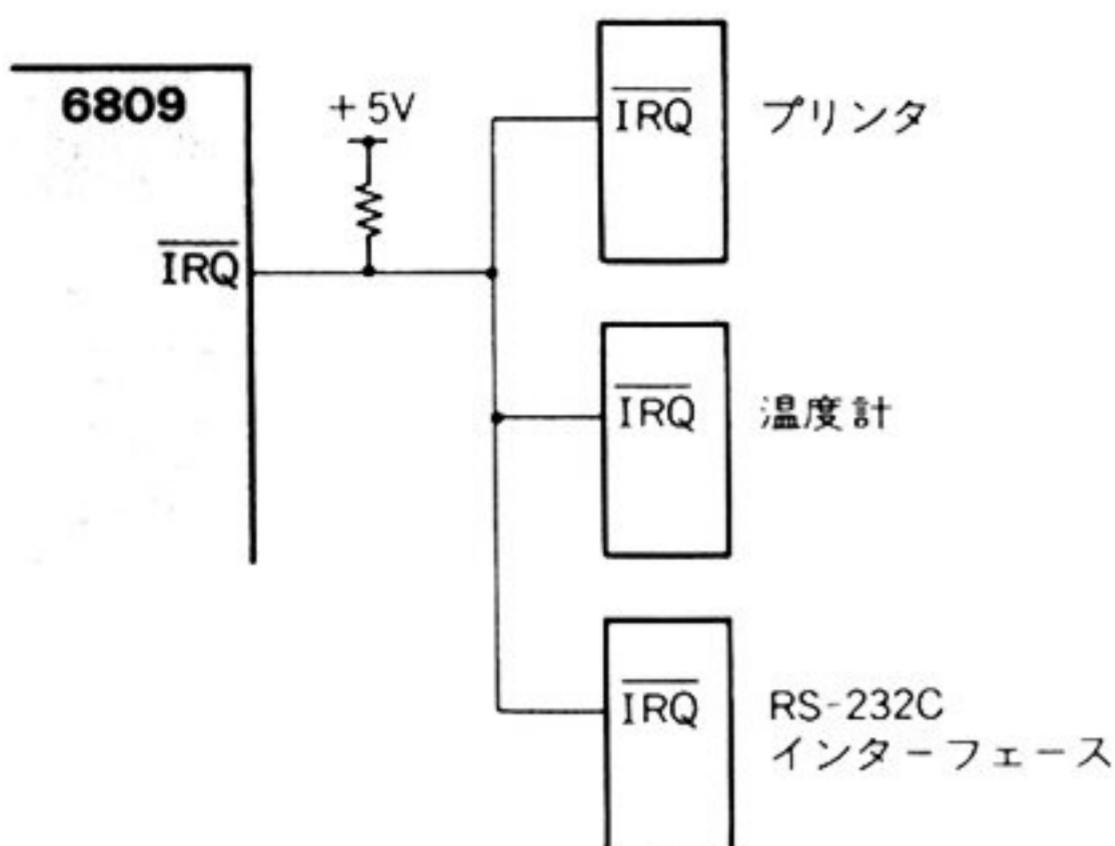



図7.5
ワイヤードORされた割り込み要求

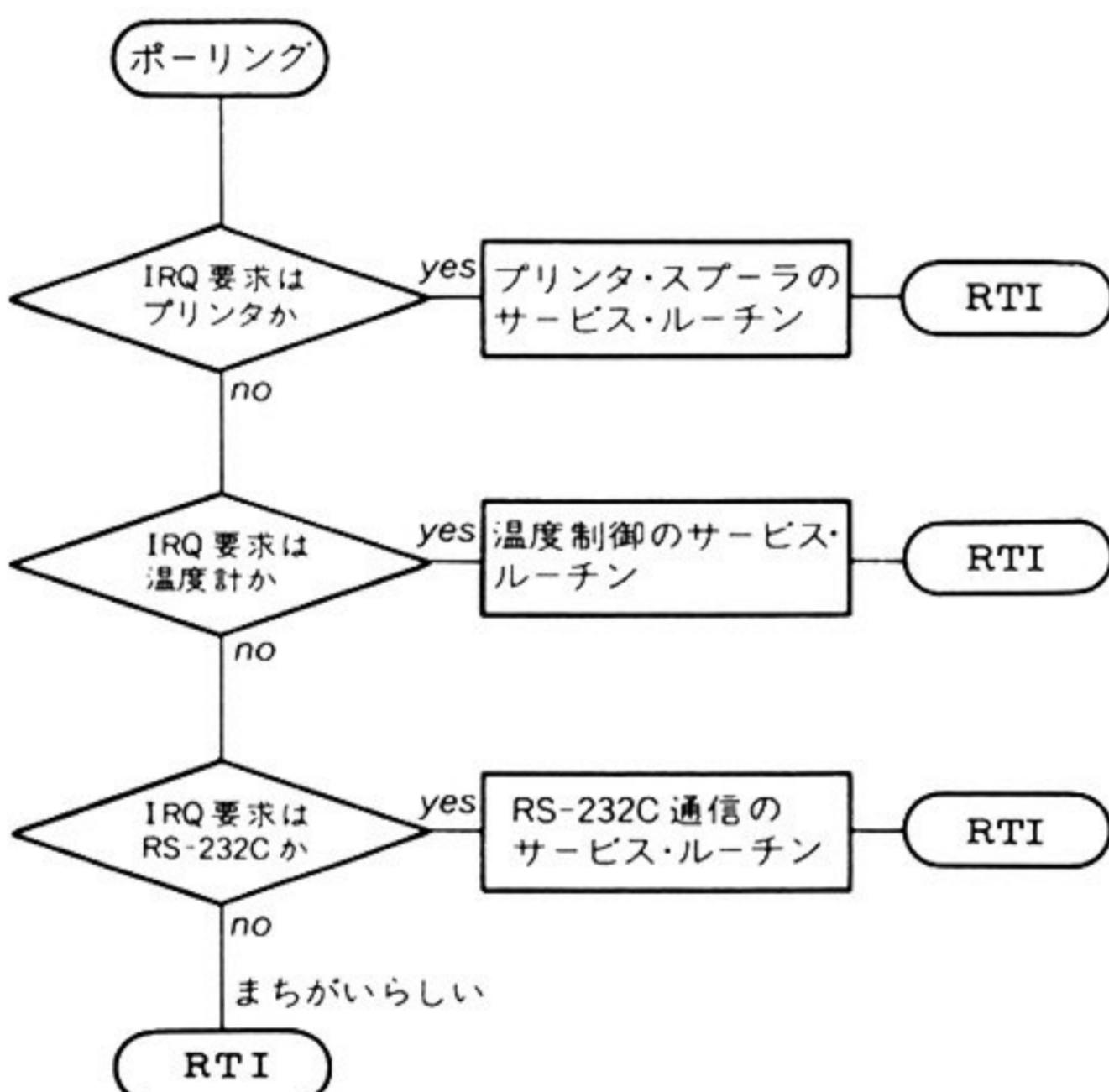



図7.6　ポーリング・ルーチン

ポーリングと呼ばれるプログラムです．

これは特別なものではなく，ポートの割り込みフラグを端から順に読みながら，割り込みを要求したのは君か，それとも君か，という具合に聞いてまわることです．

図7.5に，プリンタ，温度計それにRS-232Cインターフェースの$\overline{\mathrm{IRQ}}$がワイヤードORされてMPUに入力されているとします．これで$\overline{\mathrm{IRQ}}$が発生すれば，MPUはIRQベクタのアドレスで示される割り込みサービス・ルーチンを実行するわけですが，この時点では，

どのデバイスが割り込みを要求したのかわかりません．

そこでサービス・ルーチンでは，まず最初にポーリング・ルーチンを実行して，しかるべきサービス・ルーチンへジャンプします．このフローチャートを図7.6に示しておきました．

## 7.4　マルチ・タスク・モニタ

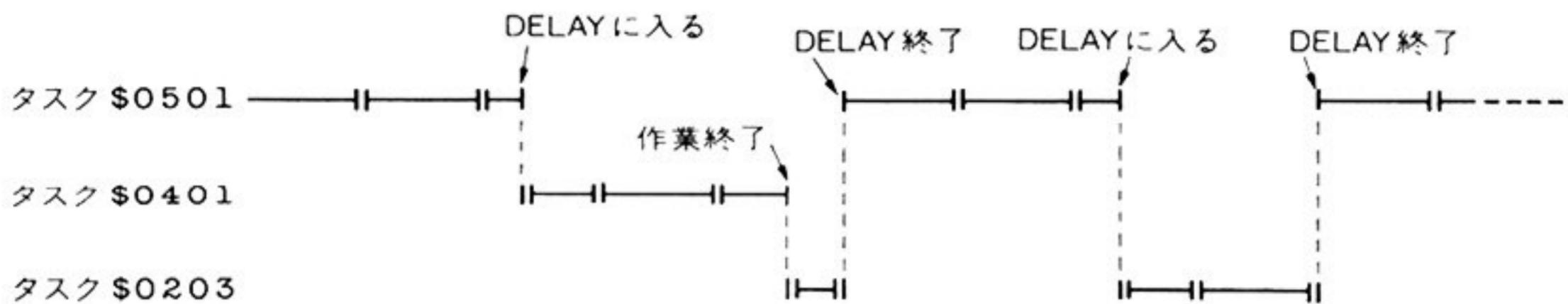
これまでに説明した多重処理は，そのほんの一部に過ぎませんが，次にマルチ・タスク・モニタ(またはリアルタイム・モニタ，RMS)を紹介します．これまでの多重処理の方法とマルチ・タスク・モニタを利用した多重処理とでは，少し概念の違いがあります．話が少し脱線しますが，このことについて数学の場合と比較してみたいと思います．

昔の小学校では，四則応用問題の解法として，「鶴亀算」「植木算」「和差算」「旅人算」「流水算」などと呼ばれる算法についてきびしい訓練を受けたのだそうです．学んだことはなくても，どなたも名前ぐらいは知っていると思います．

ところが，その生徒たちが中学校へ入ると代数を学び，方程式の使い方を教わると，それまでに大変な苦労をして解いていた問題が，何ら頭を使うことなく，機械的に解けてしまうので，なぜあんな苦労をしなくてはならなかったのかと何とも割り切れない気持ちになったのだそうです．

少し乱暴かも知れませんが，これを多重処理と比較すれば，これまでに説明したものは鶴亀算や植木算の方法に当たり，マルチ・タスク・モニタを利用する方法は代数学による手段と考えてもよいように思います．

すなわち，マルチ・タスク・モニタが要求するいくつかのルールに従ってさえいれば，極めて容易に多重処理を実現できるということです．

### ◆ マルチ・タスク・モニタの概要

それでは，簡便なマルチ・タスク・モニタを紹介しましょう．このプログラムを**リスト7.1**(pp. 151～157)に示します．

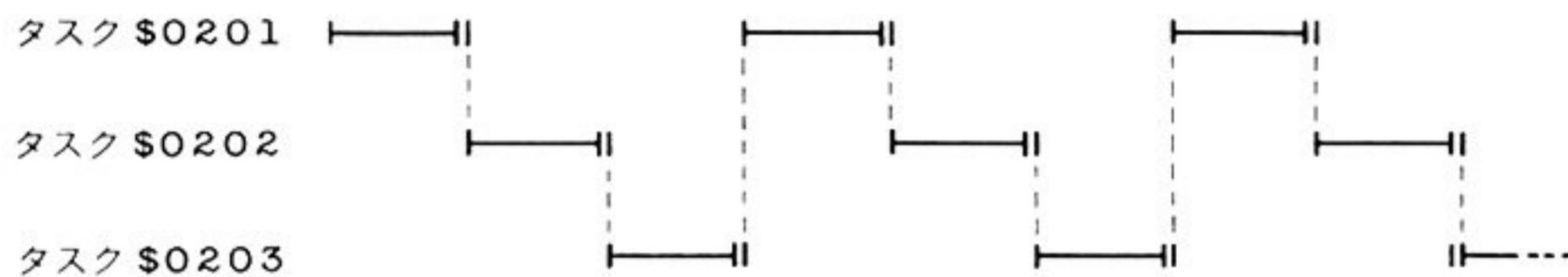
このマルチ・タスク・モニタは，6809のプログラムを効率良く，並行処理を行うためのものです．最大で15本のタスクが登録可能であり，それらのタスクを必要に応じて時分割処理または優先処理を行うことができます．

各タスクは，それぞれにタスク・ナンバが付けられ，タスク・ナンバは16ビットで構成

図7.7 複数タスクが実行される様子

(1) 優先度の異なるタスクが起動されている場合の例

(2) 優先度の同じタスクが起動されている場合の例

されます．その上位8ビットはレベル・ナンバ，下位8ビットはサブナンバとします．

それぞれのナンバは1から127まで自由に登録できますが，タスク・ナンバ0はモニタ自身のタスクであり，使用できません．

レベル・ナンバは優先順位をもち，大きな数のレベル・ナンバは高い優先度をもちます．

優先度(レベル・ナンバ)の異なるタスクが，複数同時に起動されている場合は，最も高いレベルのタスクが優先的に実行され，低いレベルのタスクは，高いレベルのタスクの空き時間(DELAY中など)を利用して実行します．

同レベルのタスクは平等に時分割で実行され，10 msごとにタスクの切り替えが行われます．この様子を図7.7に示します．

サブナンバには優先順位はなく，タスクを区別することだけに使用します．

このマルチ・タスク・モニタでは，パラメータの受け渡しにはUスタックを使用しています．これは筆者がFORTHをプログラム言語として多く利用しているため，FORTHとのパラメータの授受を容易に行うためです．

また，特定な資源の管理は行っていません．必要であれば，ユーザ・プログラムの中で行ってください．マルチ・タスクでいう資源とは，プリンタや記憶装置などのハードウェアだけでなく，複数のタスクから呼ばれる可能性のあるサブルーチンなどのプログラムも含みます．

これらの資源が，複数のタスクによって同じ時間帯に使用されては支障が生じる場合は，この使用を整理する必要があるわけです．この問題は後に資源の共同利用ということで説明します．

### ◆ マルチ・タスク・モニタの実行に必要なハードウェア

このマルチ・タスク・モニタ自身は $\overline{\mathrm{FIRQ}}$ によって起動されます．このプログラムでは，PIA(6821)の $CA_1$ へクロックを入力する設計になっています．PIA のアドレスは，リストの 300 行の SETPIA で示されています．必要があれば，この値を変更してください．

このポートのイニシャライズは，モニタの実行開始時点で行われます．ユーザ・プログラムで行う必要はありません．

クロックの周期は 10 ms を基本としますが，2 ms 以上であれば 10 ms にこだわる必要はありません．10 ms 以外のクロックを使用した場合は，その周期が DELAY およびストップ・ウォッチ機能の単位時間となります．

### ◆ マルチ・タスク・モニタのオーバヘッド

マルチ・タスク・モニタの使用中は，モニタ自身が実行するための時間が必要です．これはユーザ・プログラムのオーバヘッドとなります．

この時間は $\overline{\mathrm{FIRQ}}$ を駆動するクロックの毎周期ごとに必要であり，クロック周期はオーバヘッドとタスクの応答時間を考え合わせ，決定されなければなりません．

オーバヘッドは，タスクの数や各タスクのステートによって異なりますが，およそ毎周期ごとに 1 ms が基本であり，タスク 1 本ごとに約 100 $\mu$s 増加します．

すなわち，オーバヘッドからすれば，クロック周期は長いほうがよいわけですが，長過ぎると各タスクが同時に実行しているようには見えなくなります．

10 ms を基準と考えたのは，シーケンス制御での利用に支障を来たさないようにとしたためであり，マグネット・リレーの動作時間に近い時間を単位時間としました．

## ◈ マルチ・タスク・モニタの使い方

### ● タスク・テーブルの準備

並行処理を行おうとするすべてのプログラムは，それぞれをタスクとして，実行を開始する以前にタスク・テーブルに登録されていなくてはなりません．タスク・テーブルは**表7.1**を参照してください．

登録とは，タスクの情報をこのテーブルで示すアドレスに書き込むことですが，最低必要なことは次のとおりです．

タスク・ナンバ：上位，下位の順に1バイトずつ書き込みます．上位はレベル・ナンバであり，優先度を示します．

ステータス・フラグ：タスクの状態を示すフラグであり，ビット6と7だけが使用されています(**表7.1**参照)．ビット0から5までは0としてください．

ビット7は起動/休止を示し，0で起動されている状態，1は休止中です．

ビット6はDELAYフラグであり，1でDELAY(時間待ち)，0で実行中を示します．

登録されるタスクの最低でもどれかの1本は，起動された状態でなくてはなりません．

PH，PL：プログラム・カウンタの上位バイトと下位バイトです．ここにはタスクの実行開始アドレスを書き込みます．

UH，UL：Uレジタの上位および下位バイトです．ユーザ・スタック・ポインタの初期値を書き込みます．スタック領域はタスクごとに割り当てます．

SH，SL：Sレジスタの上位および下位バイトです．システム・スタック・ポインタの初期値を書き込みます．Sスタック領域もタスクごとに割り当てます．

上記以外のレジスタの値については，登録の必要はありませんが，登録しておけば，その値が実行開始の初期値になります．

この登録は，タスク・テーブルで示されるユーザ・タスクの領域に順に書き込みます．テーブル先頭の1行は，システムで使用されます．最後に登録したタスクの次の行のタスク・ナンバとステータスは，$FFとしてください．

### ● イニシャライザ

タスクの実行以前に，イニシャライズのためのプログラムを実行させる必要がある場合は，イニシャライザ・テーブル(**表7.1**参照)に登録することにより，イニシャライズ・ルーチンを実行させることができます．

このテーブルには，イニシャライズ・ルーチンへのジャンプ命令を書いてください(7Exxxx)．イニシャライズ・ルーチンはRTS命令で終了してください．

**表7.1 マルチ・タスク・モニタのテーブル表（各テーブルは最大15タスクまで）**

● タスク・テーブル

| | メモリ・アドレス | タスク・ナンバ | | ステータス・フラグ | PH | PL | CC | A | B | DP | XH | XL | YH | YL | UH | UL | SH | SL |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| アイドル・タスク | EB00 | 00 | 00 | | | | | | | | | | | | | | | |
| ユーザ・タスク | EB11 | | | | | | | | | | | | | | | | | |
| | EB22 | | | | | | | | | | | | | | | | | |
| | EB33 | | | | | | | | | | | | | | | | | |
| | ⋮ | | | | | | | | | | | | | | | | | |
| | | FF | FF | FF | | | | | | | | | | | | | | |

（PH～SL：レジスタの保存）

（注）テーブルの第1行は，アイドル中に使用されるのでユーザ・タスクは14個になる．
アイドルが発生しなければ，マルチ・タスクの実行後に，この行へ別のタスクを登録してもよい．
テーブルの最後は，タスク・ナンバ，ステータスはFFとすること

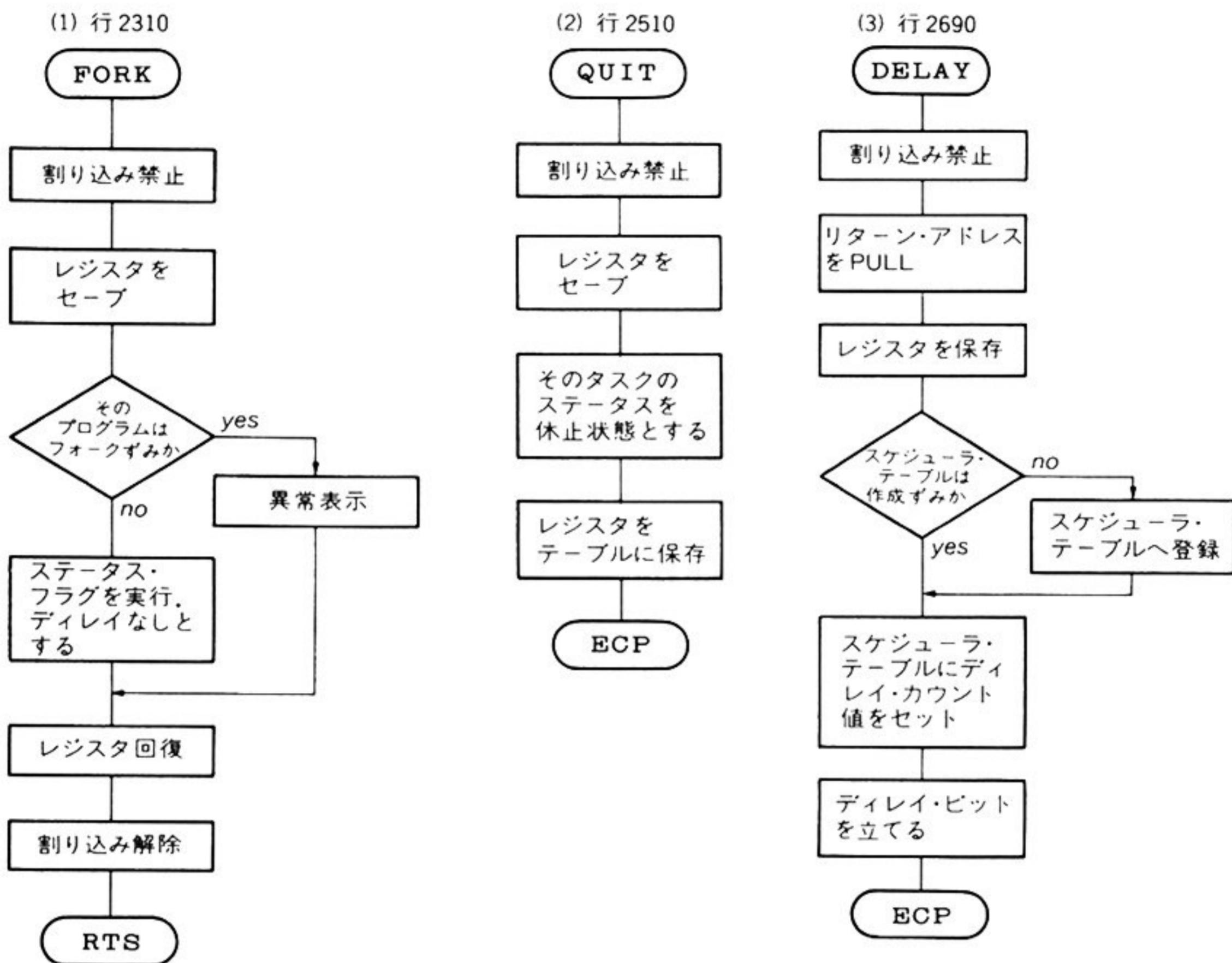
● イニシャライザ・テーブル
各3バイト

| | | | |
|---|---|---|---|
| EC00 | 7E | ×× | ×× |
| EC03 | 7E | ×× | ×× |
| EC06 | | | |
| ⋮ | | | |

RMS実行前にイニシャライズ・ルーチンを必要とする場合は，そのアドレスへのジャンプ命令を書き込む．
イニシャライズ・ルーチンは，RTSで終わること
（最大5個）

● スケジューラ・テーブル
各6バイト

| メモリ・アドレス | カウント値 | | タスク・ナンバ | | スペア | |
|---|---|---|---|---|---|---|
| EC2D | | | | | | |
| ⋮ | | | | | | |

このテーブルはユーザが意識する必要はない

● ステータス・フラグ・ビット・ナンバ

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 実行/休止 | DELAY | | | | | | |

実行＝0　休止＝1
DELAY＝1　DELAYなし＝0

図7.8　マルチ・タスク・モニタのフローチャート(1)

このイニシャライザを利用して，タスク・テーブルを作成しても構いません．

## 7.5　マルチ・タスク・モニタのサービス・ルーチン

ユーザ・プログラムがマルチ・タスク・モニタの機能を利用してタスクの制御を行えるように，次に示す五つサブルーチンが用意されています．

これらはいずれも，起動中のタスクからサブルーチン・コールして利用します．コール・アドレスは，ジャンプ・テーブルとして333から337行で示されています．

FORKは，休止状態として登録されたタスクを実行状態とするために，マルチ・タスクの実行以前にコールしても意味をもちます．

### (1)　FORK(タスクの起動)

休止中のタスクを起動する場合に使用します．

起動するタスク・ナンバをUスタックでPSHし，このサブルーチンをコールしてください．どのタスクからでも，タスクの登録がすんでいる限り使用できます．

起動済みのタスクを再びFORKした場合は，ワーニング・メッセージが表示され，ダブル・フォーク・フラグ(255行)がセットされますが，そのままプログラムの実行は継続します．

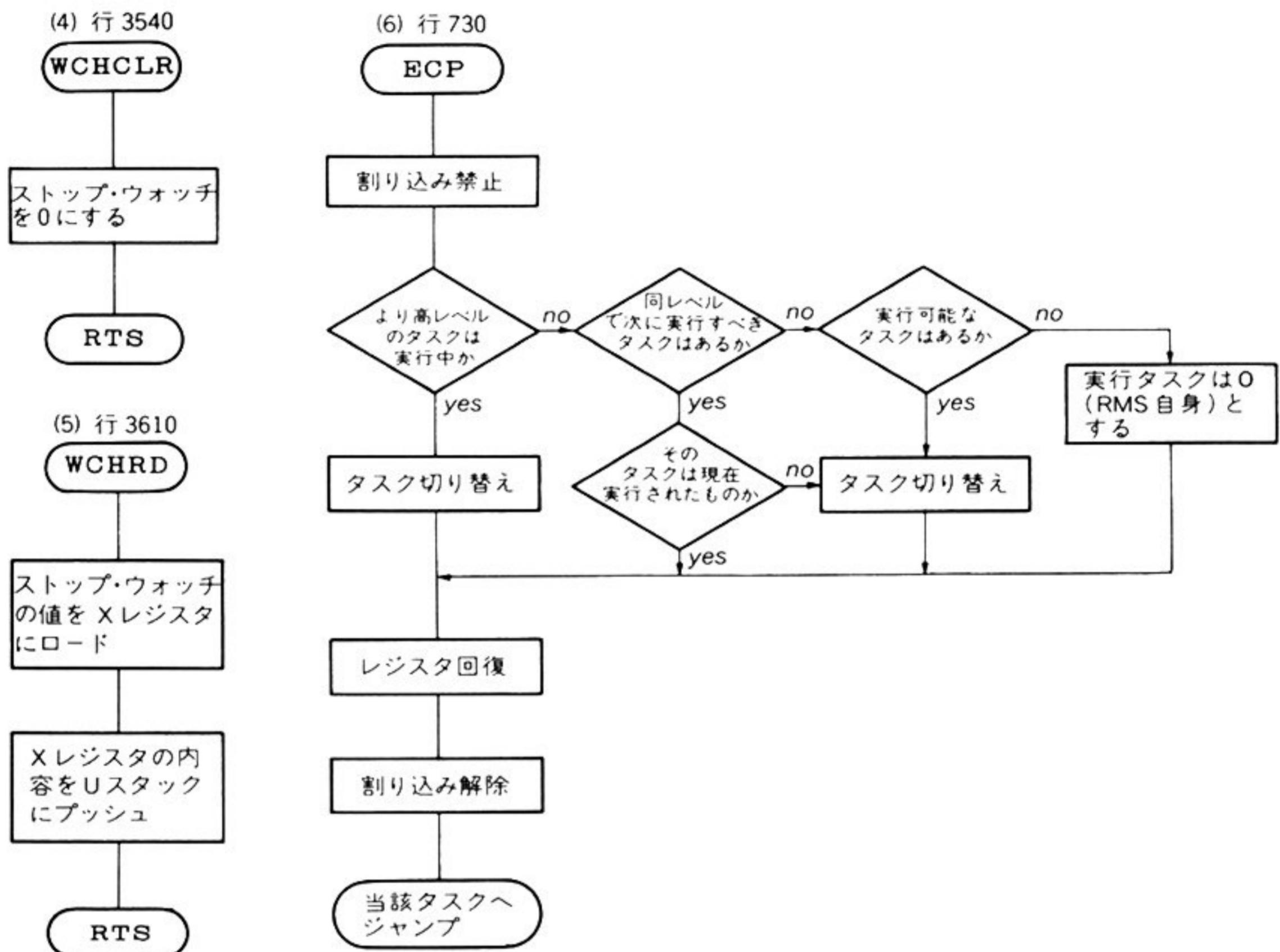
ナンバ$0305のタスクをFORKする例を以下に示します．

```
LDX     #$0305
PSHU    X
JSR     FORK
```

### (2)　QUIT(タスクの実行終了)

作業を完了したタスクは，そのタスク自身でQUITをコールしてください．パラメータはありません．QUITを行ったタスクは休止状態となり，再びほかのタスクによりFORKされた場合は，QUITを行った次の命令から実行が開始されます．

繰り返し使用するタスクは，次のようにプログラムします．

```
ラベル    ・・・・
          (実行内容)
          ・・・・
          JSR     QUIT
          JMP     ラベル
```

### (3)　DELAY(ディレイ・タイマ)

待ち時間を作るのに使用します．ディレイ時間をUスタックでPSH(16ビット)して，このサブルーチンをコールします．

時間は10 ms単位です．ディレイの最低は1 (10 ms)とします．0は$10000と判断されますので注意してください．デイレイ中は，ほかに実行可能なタスクがあれば，そのプログラムが実行されます．

マルチ・タスク・モニタ使用時には，ループ・プログラムによるソフト・タイマは正常な動作はしません．予定よりも長いディレイ時間になります．

図7.8　マルチ・タスク・モニタのフローチャート(2)

次の例は，100 msのディレイを行う例です．

```
LDD     #10
PSHU    D
JSR     DELAY
```

### (4)　WCHCLR(ストップ・ウォッチのクリア)

マルチ・タスク・モニタのもつストップ・ウォッチを0にします．パラメータはありません．

### (5)　WCHRD(ストップ・ウォッチのリード)

このマルチ・タスク・モニタでは，WCHCLRが実行されてからの時間を10 msの単位で刻んでいます．WCHRDは，WCHCLRからの経過時間を読み出し，その値を16ビッ

トでUスタックにプッシュしてリターンします．

### ◆ イベント・チェック

実行タスク以外で，周辺デバイスからの入力を常時監視するなどのイベント・チェックを必要とする場合は，そのためのサブルーチンを作成して，イベント・チェック・ルーチンとして登録しておくことができます．

この場合は，250行で示されるアドレス(EVTCK)にイベント・チェック・ルーチンへのジャンプ命令を書いてください(7Exxxx)．

このイベント・チェック・ルーチンでは，必要に応じて，FORKのコール，または，直接にタスク・テーブルのステータスを変更できます．

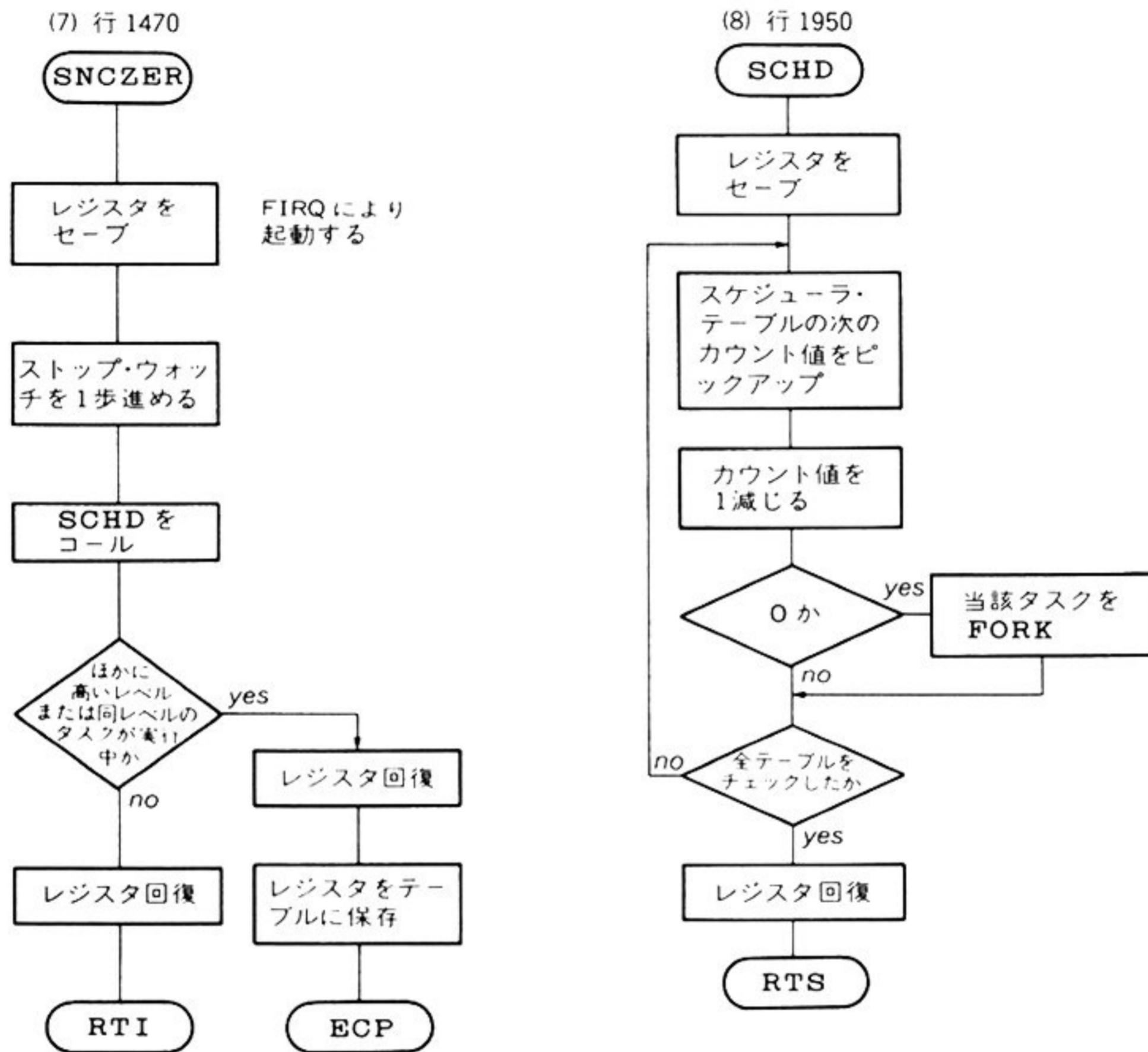
このルーチンは毎$\overline{\mathrm{FIRQ}}$クロックごとに起動され，実行中のタスクがまったくない，モニタのアイドル中にも起動されています．必要以上のオーバヘッドとならないように注意してください．

イベント・チェック・ルーチンはRTSで終了します．

## 7.6 マルチ・タスク・モニタ・プログラムの概要と実行

このマルチ・タスク・モニタは，先に述べた五つのサービス・ルーチンと，多重処理を管理実行するための三つの部分から構成されています．このフローチャートを図7.8(p.140, p.142, p.144)に示します．

このフローチャートに示した行番号は，**リスト7.1**の行番号に対応します．サービス・ルーチン以外の三つの部分について，その概要を述べておきます．

### ECP

タスク・テーブルをテストして，次に実行すべきタスクがあるかどうかを調べます．実行すべき別のタスクがあれば，タスクの切り替えを行い，そのタスクの実行を開始します．

タスクの切り替えが行われる条件は次のとおりです．

(1) 実行したタスクよりも高いレベルのタスクが実行状態になった

(2) 実行したタスク以外にも，同レベルで実行状態の別のタスクがある．

タスク・テーブルで，実行されたタスクの次の行からサーチします．

図7.8　マルチ・タスク・モニタのフローチャート(3)

(3) 実行したレベル，もしくはそれ以上のレベルのタスクはすべて休止状態または DELAY 中となったが，もっと低いレベルに実行状態のタスクがある場合

何も実行すべきタスクがない場合は，実行タスク・ナンバを0にして，アイドル状態になります．

## SNCZER

このプログラムは，$\overline{\text{FIRQ}}$によって起動されます．すなわち，10 ms ごとに起動されることになります．

ストップ・ウォッチ機能も担い，時間を示す変数 WATCH を1インクリメントします．

次に，タスク・テーブルをテストして，タスクの切り替えが必要かどうかを調べ，必要

であれば，実行したタスクのレジスタを回復し，それをタスク・テーブルに保存してECPに制御を移します．

切り替えの必要がなければ，レジスタを回復してそのままリターンして，元のプログラムを続行します．

次に述べるSCHDも，このプログラムからコールされます．

SCHD

スケジューラ・テーブル(表7.1参照)を使用して，DELAYの管理を行います．

スケジューラ・テーブルには，DELAYの実行により，待ち時間とタスク・ナンバが登録されています．このプログラムでは，DELAY中のすべてのタスクについて，待ち時間として登録されたカウント値を1デクリメントします．

カウント値が0となったタスクはこのプログラムからFORKがコールされ，実行状態にします．

### ◆ マルチ・タスク・モニタを利用した場合の待ち要素に対する処置

キー入力待ちや，フラグ待ちに対する処置は，マルチ・タスク・モニタを利用した場合にはずっと簡単になります．

最も簡単にすませるには，何の処置もいらないということです．つまり，並行して同時に実行させるタスクのレベル・ナンバをすべて同じにしておけば，平等に時分割で処理されるわけですから，プログラムの工夫だけで多重処理を行おうとした場合の，複雑な問題は起こらないわけです．

しかし，これでは少しもったいない気がします．シーケンス制御やキー操作を伴うプログラムでは，待ち要素が頻繁にあり，シンテムを利用する経過時間の90％以上はフラグ待ちに費やされることも珍しくはないからです．

そこで，これを解決する簡単な処置として，このマルチ・タスク・モニタでは，DELAYを使用することです．

キー入力を例とすれば，キーが打たれたかどうかのテストを連続して行う必要はありません．ピアニストのような指をもった人がムキになってキーをたたいたとしても，1秒間に20回もたたくことは困難でしょう．

つまり，キーが打たれたかどうかのテストは50msに1回も行ってやれば十分である

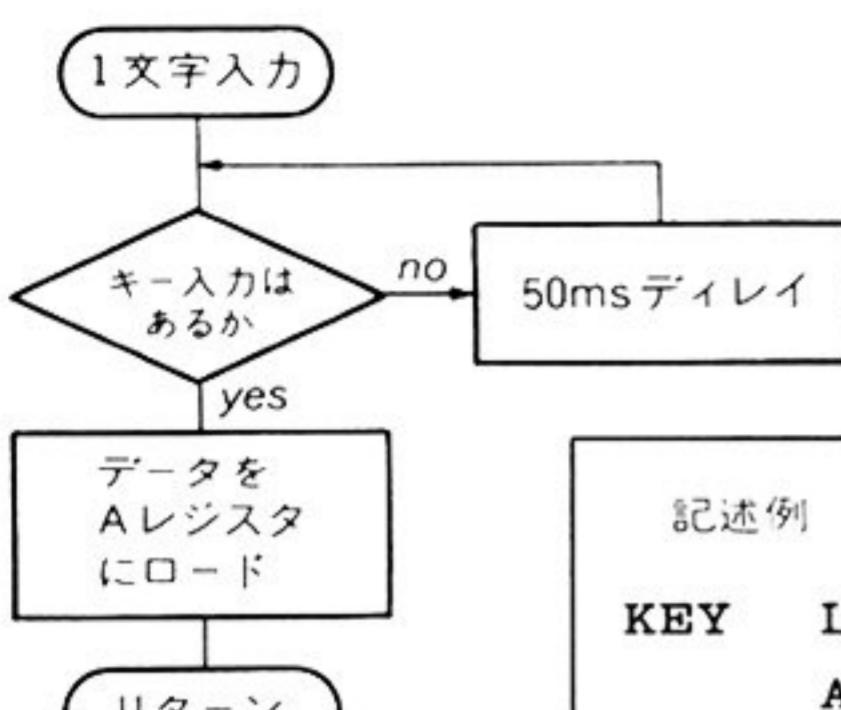
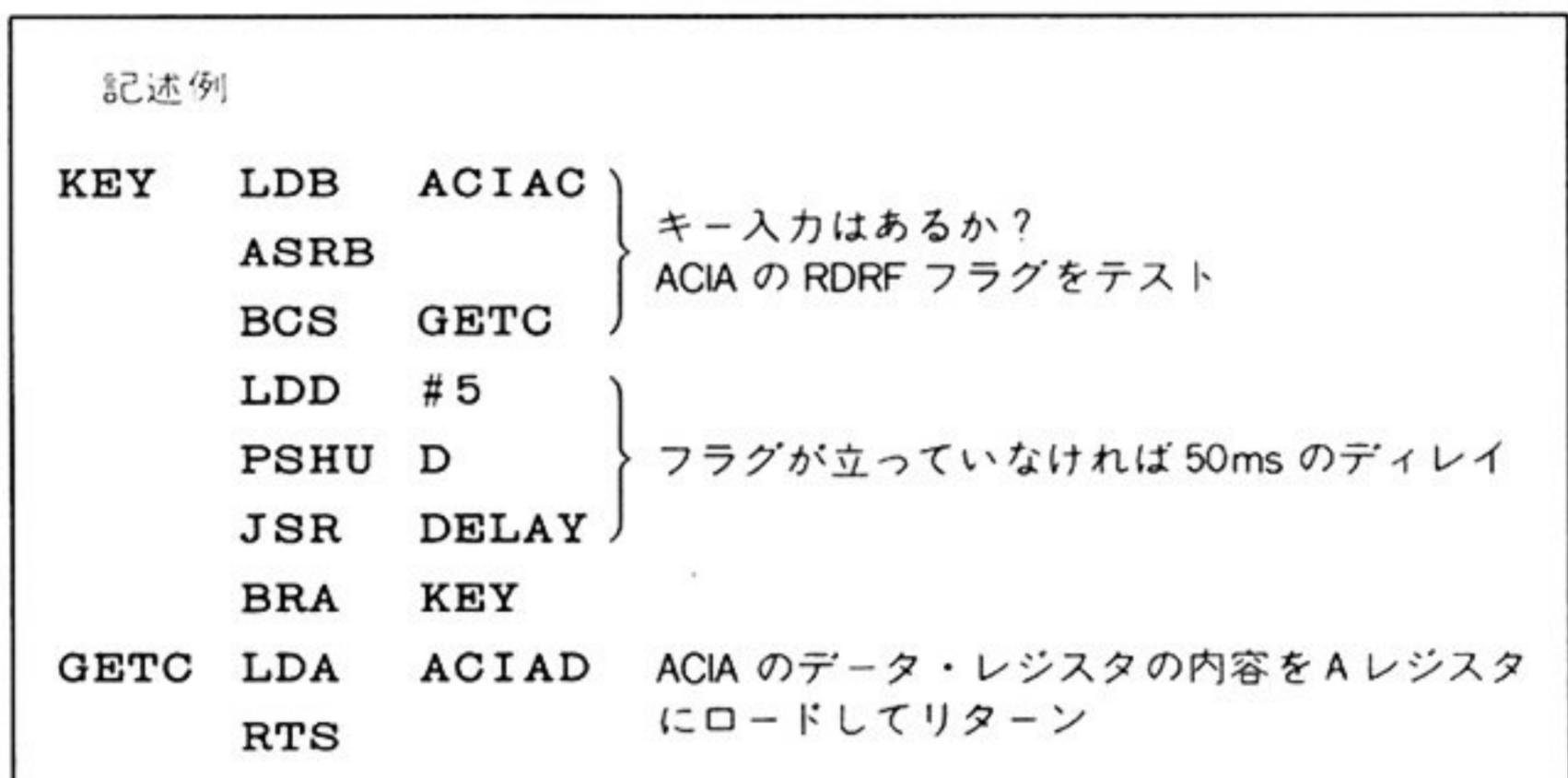



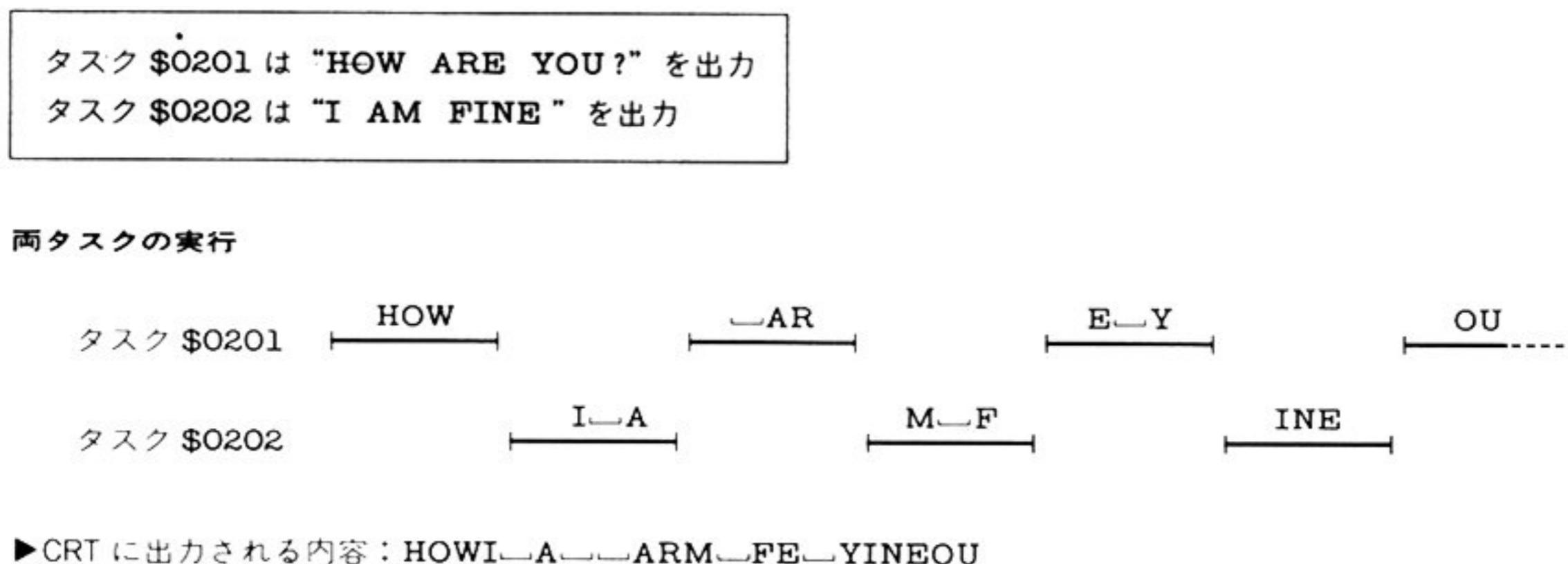
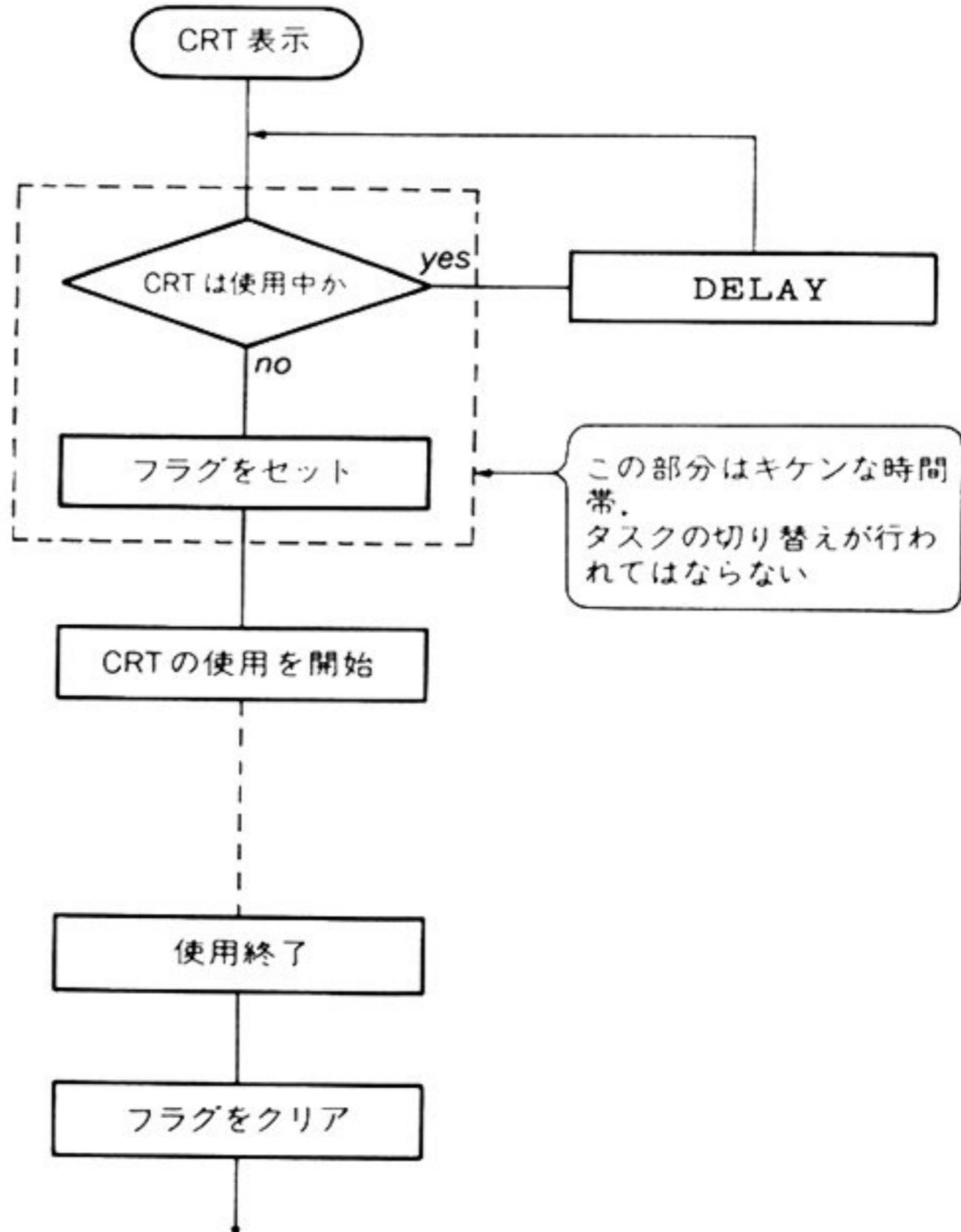
図7.9
待ち要素にDELAYを入れる
(1字入力の例)

し，それによってシステムの応答が遅いとは感じないはずです．

図7.2で示したキーからの1文字入力に，DELAYを追加したフローチャートを図7.9に示します．DELAYの使い方は，サービス・ルーチンの(3)の説明を参照してください．

DELAY中の時間はほかのタスクの実行に割り当てられるわけですから，これでシステムの利用効率がずっと良くなります．

### ◆ 資源の共同利用について

この問題について深く掘り下げることは，本書の趣旨を超えることであり，詳しい説明は省略しますが，最低必要な概念についてその概要を述べておきます．

周辺装置では，プリンタやCRT表示が身近なものなので，CRT表示を例として説明します．

今，仮に二つのタスクが時分割処理で実行中であり，両者からメッセージがCRTに表示されているとします．

この結果が図7.10で示すように，メチャクチャな表示となることは容易に想像できると思います．

このようなことが簡単に実現してしまうのもマルチ・タスクの面白いところですが，このような装置は同じ時間帯に複数のタスクが利用することはできず，なんらかの処置が必

図7.10 CRT に時分割で出力した場合

タスク $0201 は "HOW ARE YOU?" を出力
タスク $0202 は "I AM FINE " を出力

**両タスクの実行**

▶CRT に出力される内容：HOWI⌴A⌴⌴ARM⌴FE⌴YINEOU

図7.11
CRT の共同利用のための処置

要です.

この解決方法にもいろいろあるのですが，簡単なフラグの使用について述べておきます.

図7.11 に示すように，両タスク間で共通の変数をフラグとして用意しておき，CRT の使用中であることを示すようにします.

これで図7.10のような混乱はないように見えるのですが，問題はまだあり，破線で囲まれた部分の実行中にタスクの切り替えが起き，もう一方のタスクもちょうどこの部分を実行したとすれば，両者から見て，フラグはクリアされているので両方のタスクがフラグをセットしてCRT表示を開始する，つまり，図7.10と同じ結果になってしまいます．

破線の部分では，タスク切り替えが起こらなければよいわけで，68000ではテスト・アンド・セットという命令があり，これを使えば1命令ですむので問題ないのですが，6809ではどうしても複数命令となってしまいます．

このため，破線の部分でのタスク切り替えを禁止するために，破線部の先頭で，コンディション・コード・レジスタのFフラグをセット，破線部の最後でFフラグをクリアしておきます．

つまり，破線部では，$\overline{\text{FIRQ}}$に対する応答を禁止しておくのです．$\overline{\text{FIRQ}}$が禁止されれば，タスクの切り替えは起こらなくなります．

このような部分は，危険な時間帯と呼ばれています．関数サブルーチンなどのプログラムについても同様なことがいえます．

この場合は，同時に複数のタスクが使用可能かどうかは，リエントラントが可能かどうかということで議論されている問題です．

サブルーチンが，特定のメモリを作業用の変数としていれば，同時利用は不可です．作業中に中断して，別のタスクがこれを利用したとすれば，その変数は書き換えられ，元のタスクから見れば，その変数は意味をもたなくなってしまうからです．

サブルーチンで，作業用の変数が必要であれば，システム・スタック領域にローカル変数として割り当てればリエントラント可能な構造となり，そのサブルーチンは複数のタスクによって同時に利用できるようになります．

すなわち，このようなローカル変数は，サブルーチンの実行によってシステム・スタック領域に生成されるため，サブルーチンの実行が終了する以前に別のタスクが同じサブルーチンをコールしても，また新しくそのタスクのためにローカル変数が生成されます．つまり，同時に何箇所からコールされても，その数だけローカル変数が生成され，終了すれば解除されるため，互いのデータがまぜこぜになることはありません．

### ◆ マルチ・タスクを起動する手順

マルチ・タスク・モニタを利用して，複数のプログラム(タスク)を実行に移す手順は，これまでの説明の中で断片的に示されていますが，少しわかりにくいので，整理しておき

ます．

(1) すべてのタスクを単独で実行させて，デバッグを行っておく

(2) アセンブルされたマルチ・タスク・モニタのオブジェクト・コードをメモリに配置する

(3) タスクをタスク・テーブルに登録する．最低一つのタスクは実行状態であること

(4) マルチ・タスク・モニタの先頭番地(320行のORGで示されるアドレス)にジャンプする．

これで多重処理が開始するはずです．

うまく実行しない場合は，マルチ・タスク・モニタの移植をもう一度確かめてください．

2箇所のORG定義以外で，システムに依存する箇所は，**リスト7.1**の270から300行までの次に示す四つのEQU定義です．

**MON**

システム・モニタの実行開始アドレスです．

**PCLDAT**

コンソールに文字列を出力するサブルーチンのアドレス，**リスト5.5**のBUFOUTと同じ内容です．

**FIRVEC**

システム・モニタがサポートする，FIRQのベクタを格納するRAMアドレスです．

ASSIST09を使用している場合は，ベクタ・スワップ・サービスを利用して，SNCZER(1470行)のアドレスをFIRQベクタとします．この場合は，470行を以下のように書き換えます．

```
470     LDA   #10
472     SWI
474     FCB   9
```

**SETPIA**

PIA(6821)が割り付けられたアドレスです．この$CA_1$に10 msのクロックを入力して，FIRQを発行します．PIAの$\overline{IRQA}$はMPUの$\overline{FIRQ}$に配線されていることも確認してください．

## ◆ マルチ・タスクのサンプル・プログラム

簡単なプログラムを作って，このマルチ・タスク・モニタによって，実行させてみましょう．特別なハードウェアの用意を必要としないように，コンソールとプリンタだけを使ってプログラムしてみました．リスト7.2がそのサンプル・プログラムです．

TASK1とTASK2は，簡単な無限ループの独立したプログラムになっています．

ターミナルとプリンタへの1文字出力は，第5章で示したサブルーチンのOUTCHとPRNTEEを使用しています．

TASK1では，“A”をターミナルに出力し続けます．

TASK2は，キーボードから入力された文字をそのままプリンタに出力します．

キー入力ルーチンは第5章のINCHを使用することもできますが，図7.11で示したように，待ち要素のループにDELAYを加えてみました．

このプログラムをアセンブルして，「タスク・テーブルの準備」で説明したように，タスク・テーブルに登録します．

まず，タスク・ナンバは同レベルのタスクとしてみましょう(例えば，$0201と$0202)．これで多重処理の実行が確認できたら，タスクの優先レベルを変えて試みてください．

TASK1がTASK2よりも高い優先レベルの場合には，TASK2が実行しなくなるはずです．TASK1では，待ち要素のループに対する処置が何もないため，このタスクの優先度が最も高い場合には，ほかのタスクの実行時間が与えられなくなってしまうためです．

リスト7.2 マルチ・タスク・モニタを実行するサンプル・プログラム

```
02500      4009           DELAY   EQU    $4009
02510                     **
02520 013B 86      41     TASK1   LDA    #'A      タスク1
02530 013D 17      FEFC   TASK11  LBSR   OUTCH    "A"を連続してターミナルに出力
02540 0140 20      FB             BRA    TASK11
02550                     **
02560                     **
02570 0142 8E      E010   TASK2   LDX    #ACIAC   タスク2
02580 0145 CC      0005   KEY     LDD    #5       キー入力した文字コードを
02590 0148 36      06             PSHU   D        そのままプリンタに出力
02600 014A BD      4009           JSR    DELAY
02610 014D E6      84             LDB    ,X
02620 014F 57                     ASRB
02630 0150 24      F3             BCC    KEY
02640 0152 A6      01             LDA    1,X
02650 0154 84      7F             ANDA   #$7F
02660 0156 27      ED             BEQ    KEY
02670 0158 17      FF06           LBSR   PRNTEE
02680 015B 20      E8             BRA    KEY
```

リスト7.1 マルチ・タスク・モニタ

```
00100                          NAM   09RMS
00110                          OPT   NOG
00120                          OPT   S
00130                          OPT   M
00140                   **  VER. 2.0  1984 JAN. 11
00150                   **  MULTI TASK MONITOR **
00160                   **  APRIL 18 1982   BY K. TSURUMI **
00170                   **
00180 EB00                     ORG   $EB00          RMS WORK ARIA ワーク・エリアの
00190 EB00 00FF         TCB    RMB   17*15                        先頭アドレス
00200 EBFF 000F         INZTBL RMB   3*5
00210 EC0E 005A         SCHTBL RMB   6*15
00220 EC68 0002         TCBSAV RMB   2
00230 EC6A 0002         PCSAV  RMB   2
00240 EC6C 0002         CNTSAV RMB   2
00250 EC6E 0003         EVTCK  RMB   3              EVENT CHECK IF JMP CODE
00255 EC71 0001         DBLFRK RMB   1              DOUBLE FORK FLAG
00257 EC72 0002         WATCH  RMB   2              STOP WATCH DATA
00260                   **
00270      F800         MON    EQU   $F800    システム・モニタの実行開始アドレス
00280      FD10         PCLDAT EQU   $FD10    文字列出力のサブルーチン・アドレス
00290      E706         FIRVEC EQU   $E706    FIRQのベクタ
00300      E02C         SETPIA EQU   $E02C    クロックをCA1に入力するポートのアドレス
00310                   **
00320 4000                     ORG   $4000
00331                   ** JMP TABLE          ジャンプ・テーブル
00332 4000 16    000F          LBRA  INILZR
00333 4003 16    0194          LBRA  FORK     ⎫
00334 4006 16    01B8          LBRA  QUIT     ⎪ これらの機能の利用は，このアドレスに
00335 4009 16    01D6          LBRA  DELAY    ⎬ サブルーチン・ジャンプを行う
00336 400C 16    0296          LBRA  WCHCLR   ⎪
00337 400F 16    029F          LBRA  WCHRD    ⎭
00338                   **  INITIALYZER **       イニシャライズ
00340 4012 1A    50     INILZR ORCC  #%01010000                  ⎫
00350 4014 CC    0000          LDD   #0                          ⎪
00360 4017 FD    EB00          STD   TCB                         ⎪
00370 401A B7    EB02          STA   TCB+2                       ⎪
00380 401D CC    EEC0          LDD   #$EEC0      TASK0 U STACK   ⎪ アイドル・タスク
00390 4020 FD    EB0D          STD   TCB+13                      ⎬ $0000を登録
00400 4023 CC    EEFF          LDD   #$EEFF      TASK0 S STACK   ⎪
00410 4026 FD    EB0F          STD   TCB+15                      ⎪
00420 4029 318D  0208          LEAY  HUNT,PCR                    ⎪
00430 402D 10BF  EB03          STY   TCB+3                       ⎪
00440 4031 86    D0            LDA   #$D0                        ⎪
00450 4033 B7    EB05          STA   TCB+5                       ⎭
00460 4036 308D  00BD          LEAX  SNCZER,PCR ⎫ FIRQベクタの設定
00470 403A BF    E706          STX   FIRVEC     ⎭
00480 403D 86    05            LDA   #$05
00490 403F B7    E02D          STA   SETPIA+1
00500 4042 8E    EBFF          LDX   #INZTBL       INZTASK
00510 4045 A6    84     INIZ1  LDA   ,X        イニシャライザ・テーブルに登録があれば
00520 4047 81    7E            CMPA  #$7E      そのプログラムを実行
00530 4049 26    0A            BNE   INIZ2
00540 404B 34    10            PSHS  X
00550 404D AD    84            JSR   ,X
00560 404F 35    10            PULS  X
00570 4051 30    03            LEAX  3,X
00580 4053 20    F0            BRA   INIZ1
```

リスト7.1 マルチ・タスク・モニタ（つづき）

```
00590                   **
00600 4055 8E     EB00  INIZ2  LDX   #TCB         ⎫
00610 4058 BF     EC68  INIZ5  STX   TCBSAV       ⎪
00620 405B 3088   11    INIZ4  LEAX  17,X         ⎪
00630 405E EC     84           LDD   ,X           ⎪ 最もレベルの高いタスクのテーブル・
00640 4060 1083   FFFF         CMPD  #$FFFF       ⎬ アドレスをTCBSAVに保管する
00650 4064 27     09           BEQ   INIZ3        ⎪
00660 4066 10A39FEC68          CMPD  [TCBSAV]     ⎪
00670 406B 23     EE           BLS   INIZ4        ⎪
00680 406D 20     E9           BRA   INIZ5        ⎪
00690 406F 20     00    INIZ3  BRA   ECP          ⎭
00700                   **
00710                   ** EXECUTE CONTROL PROGRAM **    ECP
00720                   **
00730 4071 1A     50    ECP    ORCC  #%01010000   SET FIRQ & IRQ MASK
00740 4073 8E     EB02         LDX   #TCB+2
00750 4076 A6     84    ECP3   LDA   ,X
00760 4078 81     FF           CMPA  #$FF
00770 407A 27     1A           BEQ   ECP1         NEXT SUBLEVEL 1段上のレベル
00780 407C 85     C0           BITA  #%11000000              のタスクをサーチ
00790 407E 27     05           BEQ   ECP2         FIND RUNNING TASK 実行フラグ
00800 4080 3088   11    ECP4   LEAX  17,X                          の立ってい
00810 4083 20     F1           BRA   ECP3                          るタスクを
00820                   **                                         見つけた
00830 4085 A6     1E    ECP2   LDA   -2,X
00840 4087 A19F   EC68         CMPA  [TCBSAV]
00850 408B 22     02           BHI   CNLVL        CHANGE LEVEL NR 実行中のタス
00860 408D 20     F1           BRA   ECP4                         クよりレベル
00870                   **                                        が高ければ実
00880 408F 30     1E    CNLVL  LEAX  -2,X                         行レベルを変
00890 4091 BF     EC68  CNLVL1 STX   TCBSAV                       える
00900 4094 20     47           BRA   ECP5         END SRCH
00910                   **
00920                   ** SRCH NEXT SUB LEVEL TASK **
00930 4096 BE     EC68  ECP1   LDX   TCBSAV
00940 4099 3088   11    ECP12  LEAX  17,X
00950 409C A6     02           LDA   2,X
00960 409E 81     FF           CMPA  #$FF   テーブルの最後か
00970 40A0 27     10           BEQ   ECP11        BACK TO TABLE BEG. 見つから
00980 40A2 85     C0           BITA  #%11000000   実行中のタスクか     ないので
00990 40A4 27     02           BEQ   ECP13                            テーブル
01000 40A6 20     F1           BRA   ECP12                            の先頭へ
01010 40A8 A6     84    ECP13  LDA   ,X                               もどる
01020 40AA A19F   EC68         CMPA  [TCBSAV]
01030 40AE 27     E1           BEQ   CNLVL1       CHNGE TASK
01040 40B0 20     E7           BRA   ECP12
01050                   **
01060                   ** BACK TO TABLE BEG. **  テーブルの先頭から実行タスクを
01070 40B2 8E     EB00  ECP11  LDX   #TCB         サーチ
01080 40B5 BC     EC68  ECP15  CMPX  TCBSAV
01090 40B8 22     15           BHI   LWRTSK       LOOK LOWER TASK
01100 40BA A6     02           LDA   2,X
01110 40BC 85     C0           BITA  #%11000000
01120 40BE 27     05           BEQ   ECP14
01130 40C0 3088   11    ECP16  LEAX  17,X
01140 40C3 20     F0           BRA   ECP15
01150 40C5 A6     84    ECP14  LDA   ,X
01160 40C7 A19F   EC68         CMPA  [TCBSAV]
```

リスト7.1 マルチ・タスク・モニタ（つづき）

```
01170 40CB 27     C4             BEQ    CNLVL1
01180 40CD 20     F1             BRA    ECP16
01190                   **
01200                   ** SRCH ANY LOWER TASK **  実行レベルではタスクがないので
01210 40CF 8E     EB02 LWRTSK LDX    #TCB+2        レベルを下げて，実行可能なタス
01220 40D2 A6     84   LWTSK1 LDA    ,X            クを探す
01230 40D4 85     C0             BITA   #%11000000
01240 40D6 27     B7             BEQ    CNLVL
01250 40D8 3088   11             LEAX   17,X
01260 40DB 20     F5             BRA    LWTSK1
01270                   **
01280                   ** RESTORE REGISTERS **  タスク・テーブルからレジスタを回復
01290 40DD 10FE   EC68 ECP5      LDS    TCBSAV
01300 40E1 AE     63             LDX    3,S            PICK PC
01310 40E3 A6     65             LDA    5,S            PICK CC
01320 40E5 84     AF             ANDA   #%10101111
01330 40E7 EE     6F             LDU    15,S
01340 40E9 36     12             PSHU   X,A
01350 40EB 32     66             LEAS   6,S
01360 40ED 35     7E             PULS   D,DP,X,Y,U
01370 40EF 10EE   E4             LDS    ,S
01380 40F2 32     7D             LEAS   -3,S
01390 40F4 35     81             PULS   CC,PC          GO TO TARGET TASK
01400 40F6 12                    NOP
01430                   **
01440                   **
01450                   **  SYNCHRONIZER **        このルーチンがFIRQによって
01460                   *   ACTIVATED BY FIRQ  *   起動する
01470 40F7 34     40   SNCZER PSHS   U
01480 40F9 34     13             PSHS   A,X,CC
01490 40FB 7D     E02C           TST    SETPIA         DUMMY READ
01492 40FE BE     EC72           LDX    WATCH          WATCH INCREMENT ストップ・
01494 4101 30     01             LEAX   1,X                            ウォッチを
01496 4103 BF     EC72           STX    WATCH                          1インクリ
01500 4106 B6     EC6E           LDA    EVTCK                          メント
01510 4109 81     7E             CMPA   #$7E           JMP CODE
01520 410B 26     03             BNE    SNC0
01530 410D BD     EC6E           JSR    EVTCK
01540 4110 8D     49   SNC0      BSR    SCHD
01550                   * SRCH HIGHER OR OTHER SAME LEVEL TASK *
01560 4112 BE     EC68           LDX    TCBSAV         タスクを切り替えるべきかをテスト
01570 4115 3088   11   SNC3      LEAX   17,X
01580 4118 A6     84             LDA    ,X
01590 411A 2B     0C             BMI    SNC1
01600 411C 6D     02             TST    2,X
01610 411E 26     06             BNE    SNC6
01620 4120 A19F   EC68           CMPA   [TCBSAV]       ほかに実行すべきタスクを見つけたら
01630 4124 24     20             BHS    SNC2           SNC2 へ
01640 4126 20     ED   SNC6      BRA    SNC3
01650                   **
01660 4128 8E     EB00 SNC1      LDX    #TCB
01670 412B BC     EC68 SNC5      CMPX   TCBSAV
01680 412E 27     11             BEQ    SNC4
01690 4130 6D     02             TST    2,X
01700 4132 26     08             BNE    SNC7
01710 4134 A6     84             LDA    ,X
01720 4136 A19F   EC68           CMPA   [TCBSAV]
01730 413A 24     0A             BHS    SNC2
```

リスト7.1 マルチ・タスク・モニタ（つづき）

```
01740 413C 3088   11    SNC7    LEAX   17,X
01750 413F 20     EA            BRA    SNC5
01760                   **
01770                   * RETURN FROM SYCHRONIZER *  ほかに実行すべきタスクがないの
01780 4141 35     13    SNC4    PULS   A,X,CC        で，割り込みサービス・ルーチンを
01790 4143 35     40            PULS   U             ここで終了して元のプログラムを
01800 4145 3B                   RTI                  続行
01810                   **
01820                   ** HAS OTHER HIGHER OR SAME LEVEL TASK **
01830 4146 FE     EC68  SNC2    LDU    TCBSAV       ほかに実行すべきタスクがあったので，
01840 4149 35     13            PULS   A,X,CC       テーブルにレジスタをセーブしてECP
01850 414B 33     4D            LEAU   13,U
01860 414D 36     3F            PSHU   Y,X,DP,D,CC
01870 414F 35     1E            PULS   X,DP,D       PUL U,CC,PC
01880 4151 ED     48            STD    8,U          STORE U REGISTER
01890 4153 AF     5E            STX    -2,U         STORE PC
01900 4155 10EF   4A            STS    10,U         STORE S
01910 4158 16     FF16          LBRA   ECP
01920                   **
01930                   **
01940                   ** SCHEDULER **             スケジューラ・テーブルの管理
01950 415B 34     72    SCHD    PSHS   X,A,Y,U
01960 415D 8E     EB02          LDX    #TCB+2
01970 4160 A6     84    SCHD2   LDA    ,X
01980 4162 81     FF            CMPA   #$FF
01990 4164 27     32            BEQ    SCHD1        RETURN
02000 4166 85     40            BITA   #%01000000 } DELAY中のタスクかテスト
02010 4168 26     05            BNE    DECCNT     }
02020 416A 3088   11    SCHD3   LEAX   17,X
02030 416D 20     F1            BRA    SCHD2
02040                   **
02050 416F 2B     F9    DECCNT  BMI    SCHD3        NON ACT TASK
02060 4171 CE     EC10          LDU    #SCHTBL+2
02070 4174 10AE   C4    DEC2    LDY    ,U
02080 4177 108C   FFFF          CMPY   #$FFFF
02090 417B 1027   00E3          LBEQ   ERR4
02100 417F 10AC   1E            CMPY   -2,X
02110 4182 27     04            BEQ    DEC1
02120 4184 33     46            LEAU   6,U
02130 4186 20     EC            BRA    DEC2
02140                   **
02150                   ** DECREMENT COUNT **       カウント値をデクリメント
02160 4188 10AE   5E    DEC1    LDY    -2,U
02170 418B 31     3F            LEAY   -1,Y         0になったタスクはDELAYを
02180 418D 27     05            BEQ    WAKE         終了して実行状態にする
02190 418F 10AF   5E            STY    -2,U
02200 4192 20     D6            BRA    SCHD3
02210                   **
02220                   ** WAKE-UP  LEVEL NR STACKED BY U *
02230 4194 8D     04    WAKE    BSR    FORK
02240 4196 20     D2            BRA    SCHD3
02250                   **
02260                   ** RETURN FROM SCHEDULER *
02270 4198 35     F2    SCHD1   PULS   X,A,Y,U,PC
02280                   **
02290                   **
02300                   ** FORK  LEVEL NR STACKED BY U *  タスクを実行状態にする
02310 419A 34     33    FORK    PSHS   X,Y,A,CC
```

リスト7.1 マルチ・タスク・モニタ (つづき)

```
02320 419C 1A    50          ORCC  #%01010000
02330 419E 37    20          PULU  Y
02340 41A0 8E    EB11        LDX   #TCB+$11
02350 41A3 6D    84   FORK2  TST   ,X
02360 41A5 102B  00AF        LBMI  ERR3          NON REGISTERED TASK
02370 41A9 10AC  84          CMPY  ,X
02380 41AC 27    05          BEQ   FORK1         FIND THE TASK
02390 41AE 3088  11          LEAX  17,X
02400 41B1 20    F0          BRA   FORK2
02410                 **
02420 41B3 6D    02   FORK1  TST   2,X           二重にFORKした場合はワーニング
02430 41B5 1027  0090        LBEQ  ERR2          ALREADY ACTIVATED   を出力
02440 41B9 86    00          LDA   #0
02450 41BB A7    02          STA   2,X
02460 41BD 35    B3   FORK3  PULS  X,Y,A,CC,PC
02470 41BF 12                NOP
02475 41C0 12                NOP
02480                 **
02490                 **
02500                 ** QUIT **                  実行終了
02510 41C1 1A    50   QUIT   ORCC  #%01010000
02520 41C3 34    40          PSHS  U             レジスタをセーブしてECP
02530 41C5 34    13          PSHS  A,X,CC
02540 41C7 86    80          LDA   #%10000000
02550 41C9 BE    EC68        LDX   TCBSAV        ステータス・フラグを休止状態とする
02560 41CC A7    02          STA   2,X                      ┐
02570 41CE 1F    13          TFR   X,U                      │
02580 41D0 35    13          PULS  A,X,CC                   │
02590 41D2 33    4D          LEAU  13,U                     │
02600 41D4 36    3F          PSHU  Y,X,DP,D,CC              │
02610 41D6 35    16          PULS  D,X           PUL U,PC   ├ レジスAのセーブ
02620 41D8 ED    48          STD   8,U           STORE U    │
02630 41DA AF    5E          STX   -2,U          STORE PC   │
02640 41DC 10EF  4A          STS   10,U          STORE S    │
02650 41DF 16    FE8F        LBRA  ECP                      ┘
02660                 **
02670                 **
02680                 ** DELAY  DELAY COUNT STACKED BY U ** DELAYの実行
02690 41E2 1A    50   DELAY  ORCC  #%01010000
02700 41E4 34    51          PSHS  U,X,CC
02710 41E6 37    10          PULU  X
02720 41E8 BF    EC6C        STX   CNTSAV
02730 41EB FE    EC68        LDU   TCBSAV
02740 41EE 33    4D          LEAU  13,U
02750 41F0 35    11          PULS  CC,X
02760 41F2 36    3F          PSHU  Y,X,DP,D,CC
02770 41F4 35    16          PULS  D,X           PUL U,PC
02780 41F6 C3    0002        ADDD  #2
02790 41F9 ED    48          STD   8,U           STORE U
02800 41FB AF    5E          STX   -2,U          STORE PC
02810 41FD 10EF  4A          STS   10,U          STORE S
02820                 **
02830                 ** SRCH SCHTBL **           スケジューラ・テーブルをサーチ
02840 4200 8E    EC10 DELAY2 LDX   #SCHTBL+2
02850 4203 FE    EC68        LDU   TCBSAV
02860 4206 EC    84   DELAY3 LDD   ,X
02870 4208 2B    10          BMI   ADDDLY        NEW DELAY TASK   テーブルに
02880 420A 10A3  C4          CMPD  ,U                             なければ新
                                                                  しく登録
```

リスト7.1 マルチ・タスク・モニタ (つづき)

```
02890 420D 27     04        BEQ   DELAY1        FIND YOUR TABLE
02900 420F 30     06        LEAX  6,X
02910 4211 20     F3        BRA   DELAY3
02920                  **
02930 4213 FC     EC6C DELAY1 LDD  CNTSAV
02940 4216 ED     1E        STD   -2,X
02950 4218 20     12        BRA   DELAY4        SET DELAYBIT
02960                  **
02970                  ** NEW DELAY TASK **     スケジューラ・テーブルに新しく登録
02980 421A EC9F   EC68 ADDDLY LDD  [TCBSAV]
02990 421E ED     84        STD   ,X
03000 4220 FC     EC6C      LDD   CNTSAV
03010 4223 ED     1E        STD   -2,X
03020 4225 CC     FFFF      LDD   #$FFFF
03030 4228 ED     06        STD   6,X
03040 422A ED     04        STD   4,X
03050                  **
03060                  * SET DELAY BIT *        ステータス・フラグのDELAYビット
03070 422C A6     42   DELAY4 LDA  2,U          をセット
03080 422E 8A     40        ORA   #%01000000
03090 4230 A7     42        STA   2,U
03100 4232 16     FE3C      LBRA  ECP
03110                  **
03120                  **
03130                  ** JOB HUNTER **
03140 4235 1C     AF   HUNT   ANDCC #%10101111
03150 4237 12               NOP
03160 4238 12               NOP
03170 4239 12               NOP
03180 423A 8D     02        BSR   STBY
03190 423C 20     F7        BRA   HUNT
03200                  **
03210                  ** STAND-BY **
03220 423E B6     EC6E STBY   LDA   EVTCK
03230 4241 81     7E        CMPA  #$7E          JMP CODE
03240 4243 26     03        BNE   STBY1
03250 4245 BD     EC6E      JSR   EVTCK
03260 4248 39          STBY1  RTS
03270                  **
03280                  **
03290                  **  ERROR DISPLAYS **      エラー、ワーニングの表示ルーチン
03300 4249 308D   001F ERR2   LEAX  ERMES1,PCR
03310 424D BD     FD10      JSR   PCLDAT
03312 4250 86     01        LDA   #1
03314 4252 B7     EC71      STA   DBLFRK
03320 4255 16     FF65      LBRA  FORK3
03330 4258 308D   0024 ERR3   LEAX  ERMES2,PCR
03340 425C BD     FD10      JSR   PCLDAT
03350 425F 7E     F800      JMP   MON
03360 4262 308D   0030 ERR4   LEAX  ERMES3,PCR
03370 4266 BD     FD10      JSR   PCLDAT
03380 4269 7E     F800      JMP   MON
03390                  **
03400                  ** MESSAGE TABLE
03410                  **
03420 426C 41          ERMES1 FCC   /ALREADY ACTIVATED/
03430 427D 0A0D             FDB   $0A0D
```

リスト7.1 マルチ・タスク・モニタ（つづき）

```
03440 427F 04                FCB    4
03450 4280 4E         ERMES2 FCC    /NON REGISTERED TASK/
03460 4293 0A0D              FDB    $0A0D
03470 4295 04                FCB    4
03480 4296 53         ERMES3 FCC    /SCHTBL ERROR/
03490 42A2 0A0D              FDB    $0A0D
03500 42A4 04                FCB    4
03510                 **
03520                 **
03530                 ** WATCH CLEAR **              ストップ・ウォッチをクリア
03540 42A5 34    11   WCHCLR PSHS   CC,X
03550 42A7 1A    40          ORCC   #%01000000
03560 42A9 8E    0000        LDX    #0
03570 42AC BF    EC72        STX    WATCH
03580 42AF 35    91          PULS   CC,X,PC
03590                 **
03600                 ** WATCH READ **               ストップ・ウォッチのリード
03610 42B1 34    11   WCHRD  PSHS   CC,X
03620 42B3 1A    40          ORCC   #%01000000
03630 42B5 BE    EC72        LDX    WATCH
03640 42B8 36    10          PSHU   X
03650 42BA 35    91          PULS   CC,X,PC
03660                 **
03670                 **
03680                        END


 TCB     EB00,    INZTBL EBFF,    SCHTBL EC0E,    TCBSAV EC68
 PCSAV   EC6A,    CNTSAV EC6C,    EVTCK  EC6E,    DBLFRK EC71
 WATCH   EC72,    MON    F800,    PCLDAT FD10,    FIRVEC E706
 SETPIA  E02C,    INILZR 4012,    INIZ1  4045,    INIZ2  4055
 INIZ5   4058,    INIZ4  405B,    INIZ3  406F,    ECP    4071
 ECP3    4076,    ECP4   4080,    ECP2   4085,    CNLVL  408F
 CNLVL1  4091,    ECP1   4096,    ECP12  4099,    ECP13  40A8
 ECP11   40B2,    ECP15  40B5,    ECP16  40C0,    ECP14  40C5
 LWRTSK  40CF,    LWTSK1 40D2,    ECP5   40DD,    SNCZER 40F7
 SNC0    4110,    SNC3   4115,    SNC6   4126,    SNC1   4128
 SNC5    412B,    SNC7   413C,    SNC4   4141,    SNC2   4146
 SCHD    415B,    SCHD2  4160,    SCHD3  416A,    DECCNT 416F
 DEC2    4174,    DEC1   4188,    WAKE   4194,    SCHD1  4198
 FORK    419A,    FORK2  41A3,    FORK1  41B3,    FORK3  41BD
 QUIT    41C1,    DELAY  41E2,    DELAY2 4200,    DELAY3 4206
 DELAY1  4213,    ADDDLY 421A,    DELAY4 422C,    HUNT   4235
 STBY    423E,    STBY1  4248,    ERR2   4249,    ERR3   4258
 ERR4    4262,    ERMES1 426C,    ERMES2 4280,    ERMES3 4296
 WCHCLR  42A5,    WCHRD  42B1,

TOTAL ERRORS   00000
TOTAL WARNINGS 00000
```

### 割り込み源のクリア

周辺からの割り込み要求信号により，割り込みサービス・ルーチンを起動する場合には，そのサービス・ルーチンの実行が終了する以前に，割り込み源をクリアしなくてはなりません．

このクリアがないと，サービス・ルーチンの終了後，次の割り込みが引き続いて発生したと判断して，割り込みサービス・ルーチンが余計に起動してしまいます．

パラレル・ポートの6821の場合では，割り込み入力はポートの内部でラッチされ，そのクリアは，6821のデータ・レジスタをリードすることで行われます．すなわち，データ・レジスタを読む必要がない場合でも，割り込み源のクリアのために，サービス・ルーチン内のどこかで，6821のデータ・レジスタをダミー・リードすることになります．

# 第8章

# 6809の演算プログラム

6809では，16ビット長の解を得る加減算と乗算命令があり，簡単な制御では，特別に演算ルーチンを用意しなくてもなんとかすみますが，分解能の高いXYテーブルの制御や計測では桁数が不足することがあります．

このような場合のためと，演算プログラムの入門として4バイト長整数の四則演算，そして数表を利用した三角関数のプログラムを紹介しておきます．

このプログラムはUスタック上で演算を行い，パラメータの受渡しもUスタックを使用します．このため，必要なデータをすべてUスタックにプッシュして演算ルーチンをコールします．

演算の実行後は，結果のみがUスタックに残されています．

レジスタの保存はされていないので，必要があれば，書き加えるか，またはコール・ルーチンで行ってください．XレジスタとYレジスタの内容を乗算して，元のレジスタの内容も保存する場合の例を以下に示しておきます．

```
PSHS   X, Y
PSHU   X, Y
JSR    MUL4
PULS   X, Y
```

上の例の第1行はXとYレジスタの退避であり，4行ではそれらを回復します．2行では乗算データをUスタックに積みます．乗算の結果はUスタックに残され，Uレジスタの示すアドレスを最上位桁とする4バイトで示されます．

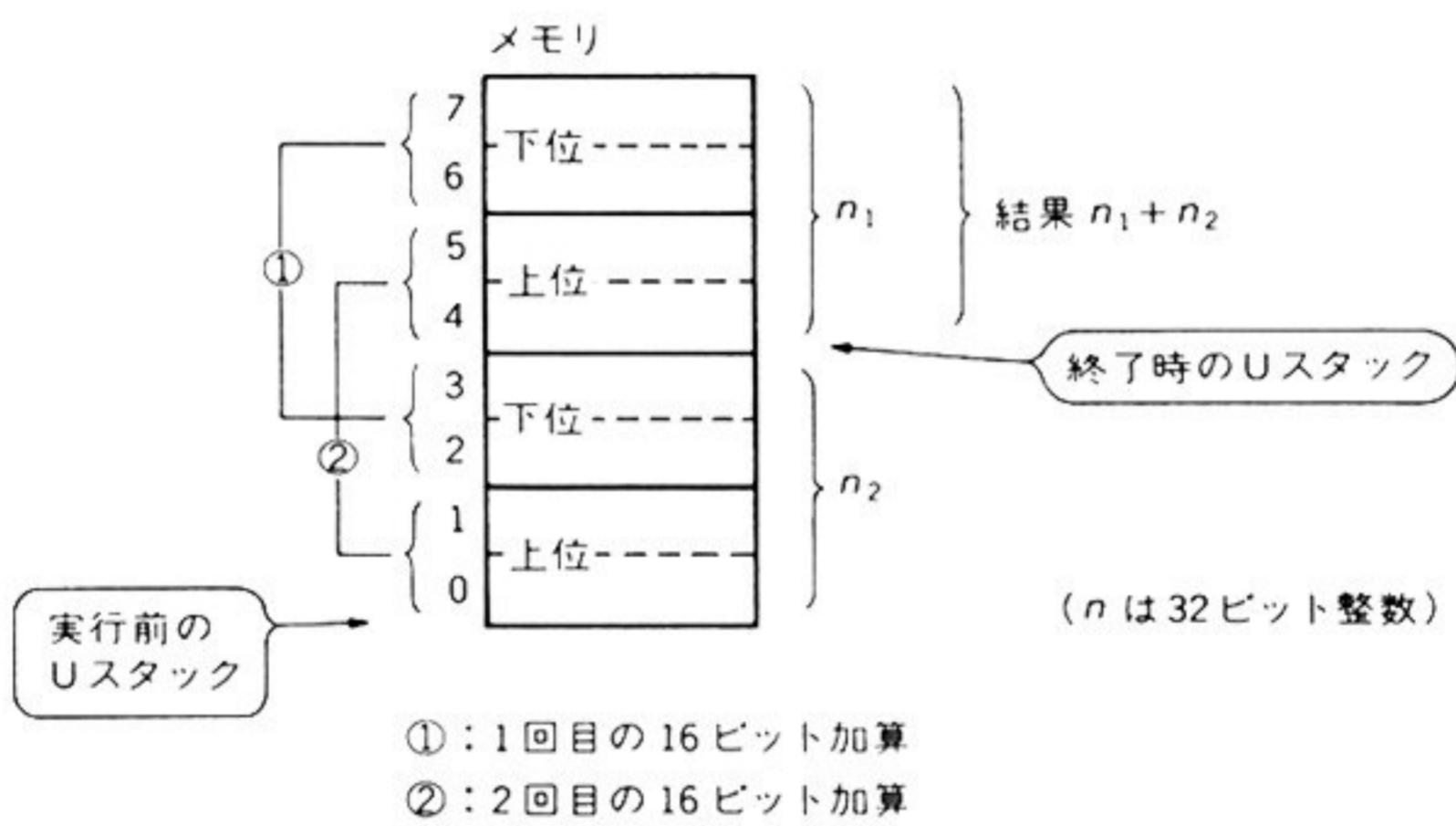
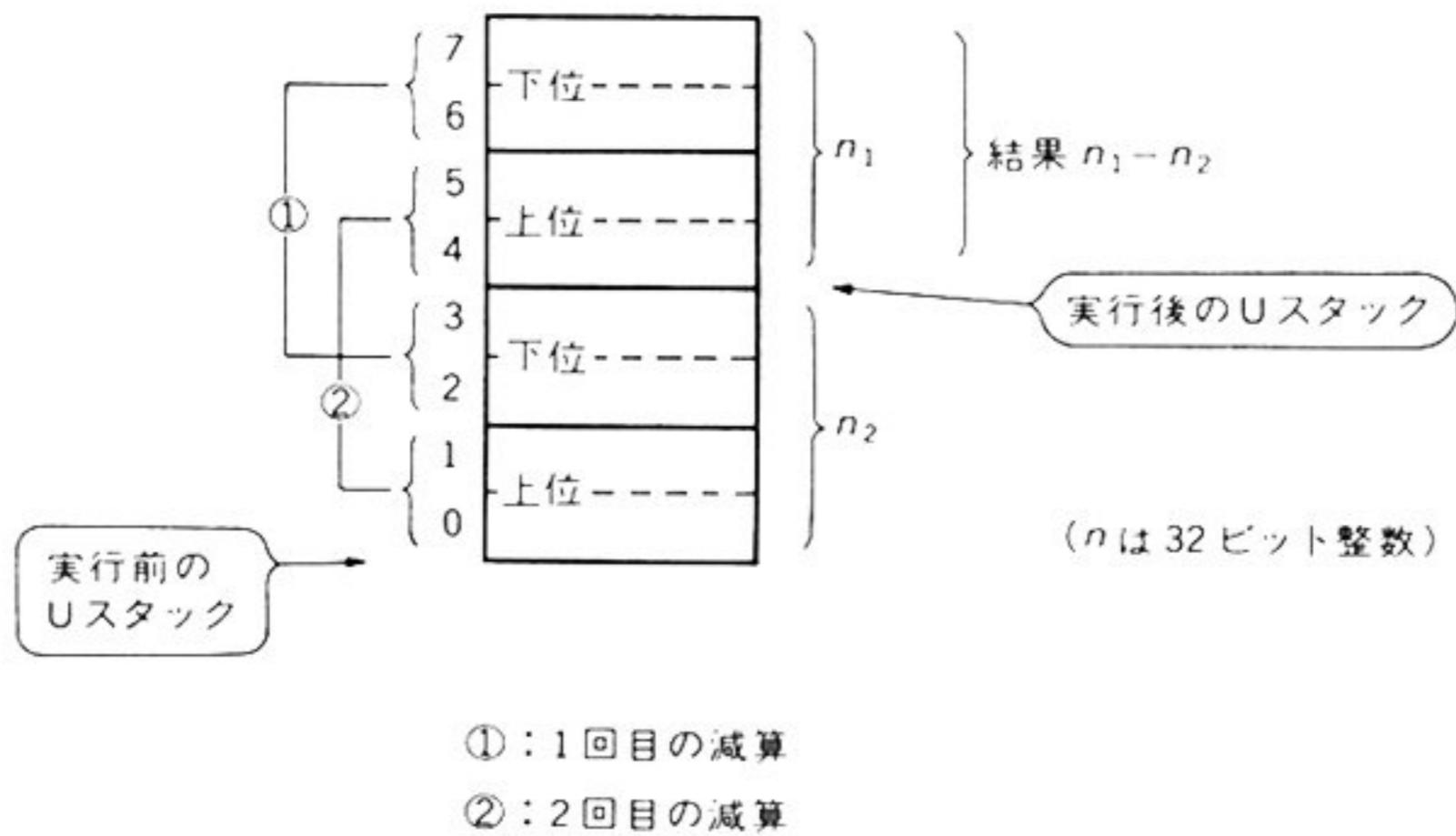
リスト8.1　演算プログラム（加算）

```
00100                         NAM   OPE.32BIT
00110                         OPT   S
00120                         OPT   M
00130                   **
00135 4000                    ORG   $4000
00140                   **
00150 4000 EC     42    ADD4  LDD   2,U  ┐
00160 4002 E3     46          ADDD  6,U  ├ ①
00170 4004 ED     46          STD   6,U  ┘
00180 4006 EC     C4          LDD   ,U   ┐   下位バイトは①の結果のキャリを含めて加算．
00190 4008 E9     45          ADCB  5,U  ├ ② 上位バイトは下位バイトのキャリを含めて加算
00200 400A A9     44          ADCA  4,U  ┘
00210 400C 33     44          LEAU  4,U
00220 400E ED     C4          STD   ,U     ②の結果をストア       図8.1の①，②参照
00230 4010 39                 RTS
00240                   **
```

図8.1　Uスタックのデータ配列（加算）

## 8.1　4バイト長の四則演算

### ● 加算

32ビット長のデータを加算して，32ビット長の結果を得ます．プログラムをリスト8.1に示します．Uスタックのデータ配列を示した図8.1を参照してください．

下位の16ビットは，Dレジスタを使用した16ビットの加算命令が利用できますが，上位の16ビットについては，キャリ付きの16ビット加算命令がないため，ADCBとADCAを使用して8ビット加算を2回行っています．

**リスト8.2　演算プログラム（減算）**

```
00250                    **
00260                    **
00270 4011 EC    46      SUB4    LDD    6,U  }
00280 4013 A3    42              SUBD   2,U  } ①
00290 4015 ED    46              STD    6,U  }
00300 4017 EC    44              LDD    4,U  }
00310 4019 E2    41              SBCB   1,U  } ② 下位バイトは①のボローを含めて減算．
00320 401B A2    C4              SBCA   ,U   }    上位バイトは下位のボローを含めて減算
00330 401D 33    44              LEAU   4,U
00340 401F ED    C4              STD    ,U      上位ワードの結果をストア 図8.2の①，②参照
00350 4021 39                    RTS
00360                    **
```

**図8.2　Uスタックのデータ配列（減算）**

①：1回目の減算

②：2回目の減算

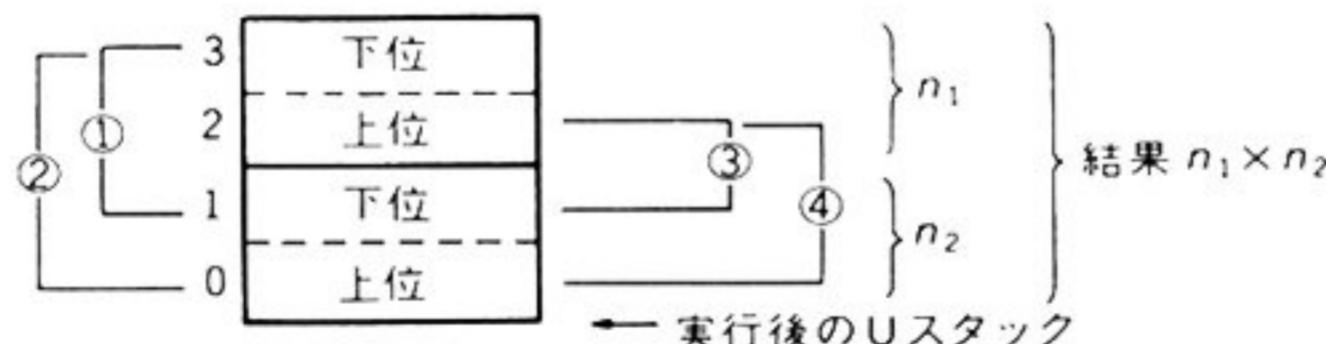
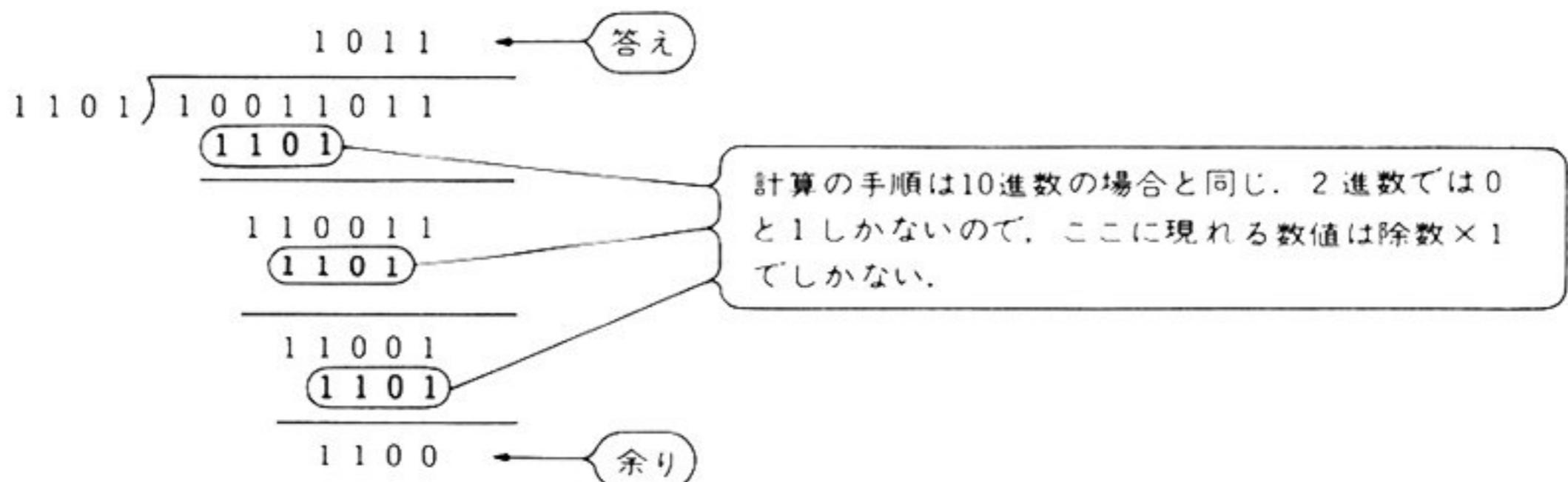
## ● 減算

32ビット長の減算を行います．最初にプッシュした数から，後からプッシュした数を引きます．

減算は，引く数を2の補数に変換して加算を行えば減算結果が得られるので，32ビットの加算プログラムを使用して簡単に作ることもできますが，機械語命令として減算命令があるので，これを利用したのが**リスト8.2**に示すサブルーチンです．

Uスタックのデータ配列は，**図8.2**に示しておきます．下位の16ビットはDレジスタによる16ビットの減算命令を使いましたが，加算の時と同様に，ボロー付きの16ビット減算命令はないため，上位の16ビットは8ビットずつ行います．

減算命令を実行した後では，キャリ・フラグはボローを意味することに注意してください．

## リスト8.3　演算プログラム（乗算）

```
00370                        **
00380                        **
00390 4022 37     30         MUL4     PULU   X,Y  } Uの2ワードをSに転送
00400 4024 34     30                  PSHS   X,Y  }
00410 4026 A6     61                  LDA    1,S  }
00420 4028 E6     63                  LDB    3,S  } ①
00430 402A 3D                         MUL         }
00440 402B 36     06                  PSHU   D
00450 402D 33     5E                  LEAU   -2,U
00460 402F 6F     C4                  CLR    ,U
00470 4031 A6     E4                  LDA    ,S   }
00480 4033 E6     63                  LDB    3,S  } ②
00490 4035 3D                         MUL         }
00500 4036 EB     42                  ADDB   2,U  }
00510 4038 89     00                  ADCA   #0   } ②結果を①の結果に加える
00520 403A ED     41                  STD    1,U  }
00530 403C A6     61                  LDA    1,S  }
00540 403E E6     62                  LDB    2,S  } ③
00550 4040 3D                         MUL         }
00560 4041 EB     42                  ADDB   2,U  } ③の結果に②の結果を加える
00570 4043 A9     41                  ADCA   1,U  }
00580 4045 24     02                  BCC    MUL41 } キャリがセットされれば上位桁を
00590 4047 6C     C4                  INC    ,U    } 1インクリメント
00600 4049 ED     41         MUL41    STD    1,U   }
00610 404B A6     E4                  LDA    ,S   }
00620 404D E6     62                  LDB    2,S  } ④
00630 404F 3D                         MUL         }
00640 4050 EB     41                  ADDB   1,U
00650 4052 A9     C4                  ADCA   ,U                 図8.3の①,②,③,④を参照
00660 4054 ED     C4                  STD    ,U
00670 4056 32     64                  LEAS   4,S    Sスタックのローカル変数を解除
00680 4058 39                         RTS
00690                        **
```

## 図8.3　Uスタックのデータ配列（乗算）

実行前と実行後のUスタックの位置は変わらない

（$n$ は16ビット整数）

①：1回目の乗算

②：2回目の乗算

③：3回目の乗算

④：4回目の乗算

図8.4　155÷13を2進数で計算する手順

## ● 乗算(符号なし)

8ビットのマイクロプロセッサで乗算プログラムを作る場合には，右シフトと加算を繰り返す方法が一般的ですが，6809では8ビットと8ビットの乗算を行い，16ビットの結果を得るMUL命令があります．これを利用して，部分積の和を得る方法で，16ビットと16ビットの乗算を行い32ビットの結果を得るプログラムを紹介します．

実行時間も，シフトと加算を繰り返す方法よりもずっと短くなります．プログラムはリスト8.3，データの配置の様子を図8.3に示しました．

アルゴリズムは，手計算による方法と同様です．

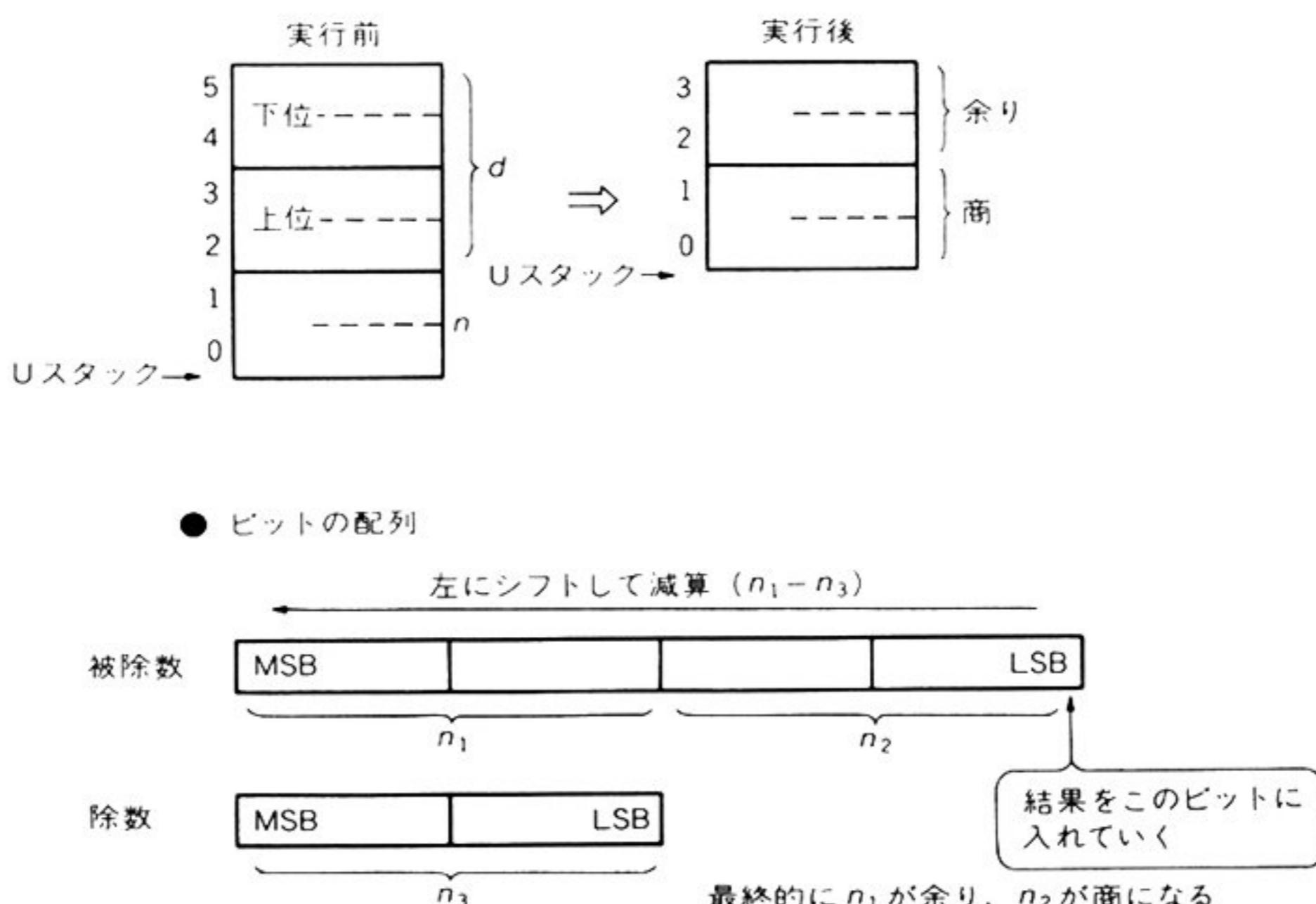
8ビットを1桁として，2桁同士の乗算と考えればよいわけですから，MUL命令を4回実行して桁を移動しながら加えます．3回目の結果を加算したときには桁上げが発生する場合があるので，その処置も必要です．

このプログラムでは，Uスタックのデータをいったん S スタックに移し，S スタックに一時的なローカル変数を割り付けた形で計算を進めています．

## ● 除算(符号なし)

さすがの6809も除算命令はもっていません．シフトと減算命令を使用して作らなくてはなりません．

考え方は手計算の場合と同じであり，これを2進数で行うことを考えればよいわけです．155/13を2進数で計算した例を図8.4に示しておきます．

この図から，マイクロプロセッサで同じことを実行させるには，被除数を左にシフトしながら除数で減算を試み，減算ができたら答えのビットに1を立てる，という具合にして上位のビットから1ビットずつ結果を得ればよいことがわかります．

### リスト8.4　演算プログラム（除算）

```
00700                        **
00710                        **
00720  4059  EC    42        DIV4    LDD    2,U   ┐
00730  405B  AE    44                LDX    4,U   │ 被除数の上位ワードと下位ワードを入れ換える
00740  405D  AF    42                STX    2,U   │
00750  405F  ED    44                STD    4,U   ┘
00760  4061  68    43                ASL    3,U   ┐ 被除数の下位を左シフト
00770  4063  69    42                ROL    2,U   ┘
00780  4065  8E    0010              LDX    #$10    回数の初期値
00790  4068  69    45        DIV41   ROL    5,U   ┐ 被除数の上位を左シフト
00800  406A  69    44                ROL    4,U   ┘
00810  406C  EC    44                LDD    4,U   ┐ 減算
00820  406E  A3    C4                SUBD   ,U    ┘
00830  4070  1C    FE                ANDCC  #$FE    キャリ・フラグをリセット
00840  4072  2B    04                BMI    DIV42 ┐ 減算可能であれば被除数を減算する
00850  4074  ED    44                STD    4,U   ┘
00860  4076  1A    01                ORCC   #1      結果の1ビット
00870  4078  69    43        DIV42   ROL    3,U   ┐ 下位ワードを左シフト
00880  407A  69    42                ROL    2,U   ┘
00890  407C  30    1F                LEAX   -1,X  ┐
00900  407E  26    E8                BNE    DIV41 │ 16回行ったら終了
00910  4080  33    42                LEAU   2,U   ┘
00920  4082  39                      RTS
00930                        **
00940                        **
```

### 図8.5　Uスタックのデータ配列（除算）とビットの配列

31ビットを16ビットで割り，16ビットの商と16ビットの余りを得るプログラムを**リスト8.4**に示します．Uスタックでのデータの配置は**図8.5**に示しました．

## 8.2　数表により三角関数を求める

三角関数や指数関数を計算で求めるには，テイラー展開があまりにも有名であり，それ以外にも収束の速いものや計算機の性格に適した近似公式がたくさんあります．

実数演算を高速で実行する用意があれば，このような近似式を利用すべきですが，その用意がない場合にも，いつも同じ方法を適用させようとするのはあまり得策ではないと思います．

そこで，少し古くさい話になりますが，数表を利用する方法を考えてみましょう．

一昔前までは，初等関数を含む式を実際に計算するには，三角関数表や対数表といった数表を利用しながら計算を進めたものです．

しかし，キー操作一発で8桁以上の値を得る関数電卓が手軽に使えるようになってからは，数表もその価値が薄れてしまったようです．

この数表を引く作業をプログラムでやらせようというわけですが，6809のアドレシング能力は大変に優れているので，驚くほど速く解を引き出すことができます．

数表をメモリに置くとなると，その量が膨大になると思われるかも知れませんが，少し工夫をすればたいしたものではありません．三角関数では，0から90度までの値があれば，すべての角についてsinとcosが求まるし，精度が少し犠牲になるのを覚悟すれば，sinをcosで割り，tanを求めることもできます．

**リスト8.5**に，sinとcosを求めるプログラムを示します．単位を度とした16ビットの正の数に対して，小数点以下4桁のsinとcosの値が得られます．

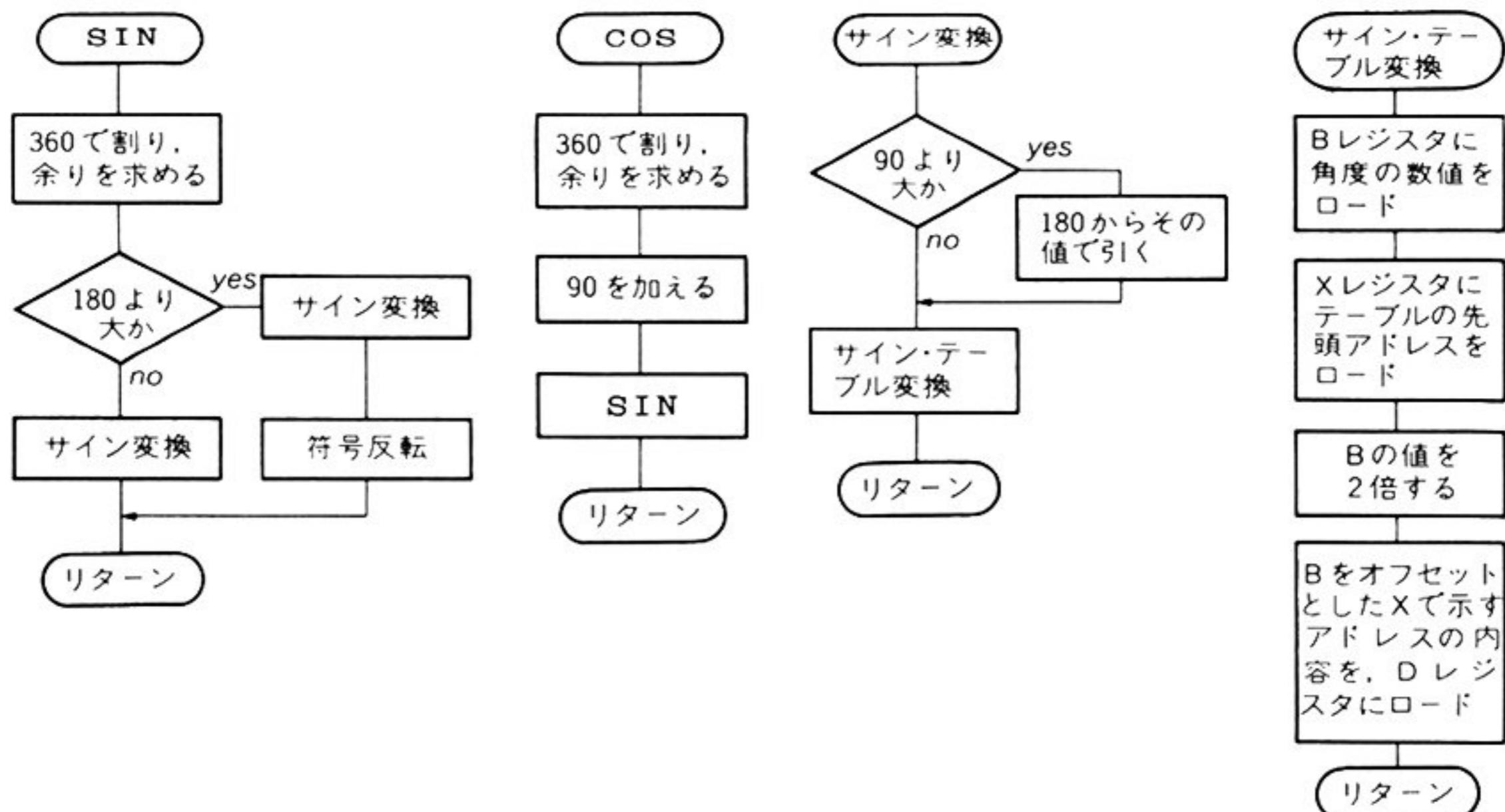



**図8.6**にフローチャートを示しておきましたので，これを参照してプログラムの仕組みは理解できると思います．実際に0から90度までの数表を引く作業をするのはSTBLCVであり，それ以外の部分では，0から65535度の範囲で与えられるsinとcosの角を0から90までのsinの角に変換しています．

はじめから0から90までの範囲のsinであれば，STBLCVを直接コールすれば，その分だけ実行時間が少なくてすみます．

このプログラムも四則演算の場合と同様に，データはUスタックで受渡しを行います．16ビット・サイズの度を単位とした数をUスタックにプッシュして，このサブルーチンを

コールします．実行後は小数点以下4桁の固定小数点で，結果のみが16ビット・サイズのデータとしてUスタックに残ります．

図8.6 sin, cos を求めるフローチャート

リスト8.5　演算プログラム（sin, cos）

```
00950                     **
00960                     **
00970 4083 4F             SIN    CLRA           サイン
00980 4084 5F                    CLRB
00990 4085 36      06            PSHU   D
01000 4087 CC      0168          LDD    #360  }
01010 408A 36      06            PSHU   D     }
01020 408C 17      FFCA          LBSR   DIV4  } 360で割り，余りのみを残す
01030 408F 33      42            LEAU   2,U   }
01040 4091 EC      C4            LDD    ,U
01050 4093 1083    00B4          CMPD   #180    180より大か
01060 4097 23      0B            BLS    SIN1
01070 4099 83      00B4          SUBD   #180
01080 409C ED      C4            STD    ,U
01090 409E 8D      1E            BSR    S180    サイン変換
01100 40A0 8D      3A            BSR    NEGD    符号反転
01110 40A2 20      02            BRA    SIN2
01120 40A4 8D      18     SIN1   BSR    S180    サイン変換
01130 40A6 39             SIN2   RTS
01140                     **
```

## リスト8.5 演算プログラム（つづき）

```
01150                    **
01160                    **
01170 40A7 4F            COS    CLRA              コサイン
01180 40A8 5F                   CLRB
01190 40A9 36    06             PSHU   D
01200 40AB CC    0168           LDD    #360   ┐
01210 40AE 36    06             PSHU   D      │ 360で割り，余りに90を加える
01220 40B0 17    FFA6           LBSR   DIV4   │
01230 40B3 CC    005A           LDD    #90    │
01240 40B6 33    42             LEAU   2,U    │
01250 40B8 E3    C4             ADDD   ,U     │
01260 40BA ED    C4             STD    ,U     ┘
01270 40BC 20    C5             BRA    SIN          サイン
01280                    **
01290                    **
01300 40BE EC    C4      S180   LDD    ,U           サイン変換
01310 40C0 1083  005A           CMPD   #90          90より大か
01320 40C4 23    07             BLS    S1801
01330 40C6 CC    00B4           LDD    #180     ┐
01340 40C9 A3    C4             SUBD   ,U       │ 180からデータで引く
01350 40CB ED    C4             STD    ,U       ┘
01360 40CD 8D    01      S1801  BSR    STBLCV       テーブル変換
01370 40CF 39                   RTS
01380                    **
01390                    **
01400                    **
01410 40D0 EC    C4      STBLCV LDD    ,U           テーブル変換
01420 40D2 308D  0010           LEAX   SINTBL,PCR
01430 40D6 58                   LSLB                Bを2倍
01440 40D7 EC    8B             LDD    D,X          Dに結果をロード
01450 40D9 ED    C4             STD    ,U           Uスタックに結果をストア
01460 40DB 39                   RTS
01470                    **
01480                    * NEGATE STACK DATA *      符号反転
01490 40DC 60    41      NEGD   NEG    1,U
01500 40DE 25    03             BCS    NEGD1
01510 40E0 60    C4             NEG    ,U
01520 40E2 39                   RTS
01530 40E3 63    C4      NEGD1  COM    ,U
01540 40E5 39                   RTS
01550                    **
01560                    **
01570                    * SINE TABLE *             サイン数表（0～90°のサインの値）
01580 40E6 0000          SINTBL FDB    0000,0175,0349,0523,0698
      40E8 00AF
      40EA 015D
      40EC 020B
      40EE 02BA
01590 40F0 0368                 FDB    0872,1045,1219,1392,1564
      40F2 0415
      40F4 04C3
      40F6 0570
      40F8 061C
01600 40FA 06C8                 FDB    1736,1908,2079,2250,2419
      40FC 0774
      40FE 081F
      4100 08CA
      4102 0973
```

### リスト8.5 演算プログラム（つづき）

```
01610 4104 0A1C          FDB     2588,2756,2924,3039,3256
      4106 0AC4
      4108 0B6C
      410A 0BDF
      410C 0CB8
01620 410E 0D5C          FDB     3420,3584,3746,3907,4067
      4110 0E00
      4112 0EA2
      4114 0F43
      4116 0FE3
01630 4118 1082          FDB     4226,4384,4540,4695,4848
      411A 1120
      411C 11BC
      411E 1257
      4120 12F0
01640 4122 1388          FDB     5000,5150,5299,5446,5592
      4124 141E
      4126 14B3
      4128 1546
      412A 15D8
01650 412C 1668          FDB     5736,5878,6018,6157,6293
      412E 16F6
      4130 1782
      4132 180D
      4134 1895
01660 4136 191C          FDB     6428,6561,6691,6820,6947
      4138 19A1
      413A 1A23
      413C 1AA4
      413E 1B23
01670 4140 1B9F          FDB     7071,7193,7314,7431,7547
      4142 1C19
      4144 1C92
      4146 1D07
      4148 1D7B
01680 414A 1DEC          FDB     7660,7771,7880,7986,8090
      414C 1E5B
      414E 1EC8
      4150 1F32
      4152 1F9A
01690 4154 2000          FDB     8192,8290,8387,8480,8572
      4156 2062
      4158 20C3
      415A 2120
      415C 217C
01700 415E 21D4          FDB     8660,8746,8829,8910,8988
      4160 222A
      4162 227D
      4164 22CE
      4166 231C
01710 4168 2367          FDB     9063,9135,9205,9272,9336
      416A 23AF
      416C 23F5
      416E 2438
      4170 2478
```

リスト8.5　演算プログラム（つづき）

```
01720 4172 24B5               FDB    9397,9455,9511,9563,9613
      4174 24EF
      4176 2527
      4178 255B
      417A 258D
01730 417C 25BB               FDB    9659,9703,9744,9781,9816
      417E 25E7
      4180 2610
      4182 2635
      4184 2658
01740 4186 2678               FDB    9848,9877,9903,9925,9945
      4188 2695
      418A 26AF
      418C 26C5
      418E 26D9
01750 4190 26EA               FDB    9962,9976,9986,9994,9998
      4192 26F8
      4194 2702
      4196 270A
      4198 270E
01760 419A 2710               FDB    10000
01770                  **
01780                  **
01790                         END


  ADD4    4000,   SUB4   4011,   MUL4   4022,   MUL41  4049
  DIV4    4059,   DIV41  4068,   DIV42  4078,   SIN    4083
  SIN1    40A4,   SIN2   40A6,   COS    40A7,   S180   40BE
  S1801   40CD,   STBLCV 40D0,   NEGD   40DC,   NEGD1  40E3
  SINTBL  40E6,

TOTAL ERRORS   00000
TOTAL WARNINGS 00000
```

## 参考・引用*文献

(1) 高橋登，中川誠治，中村惠一，清水義和；6809セルフアセンブラ，インターフェース，1981年 2月号，CQ出版社.

(2)*Motorola Microprocessors Data Manual, MOTOROLA Semiconductor Products Inc,.

(3) 加瀬 清；6809ハンドブック，アスキー.

(4) MC 6809-MC 6809 E マイクロプロセッサ プログラミング マニュアル，CQ出版社.

(5) 大川善邦；演算プログラムの作り方，産報出版.

(6)*西沢 昭；Z80上級プログラミング，CQ出版社.

(7) THE COMPLETE MOTOROLA MICROCOMPUTER DATA LIBRARY, MOTOROLA Semiconductor Products Inc,.

(8)*intel Data Catalog 1977. Intel Corporation.

(9)*日立 マイクロコンピュータ データブック，8ビット・16ビットマルチチップ，59年5月，日立製作所.

(10)*赤 攝也；数学と人間生活，日本放送出版協会.

# 索　引

## 【ア　行】



## 【カ　行】



## 【サ　行】

## 【タ　行】



## 【ナ　行】



## 【ハ　行】

【マ　行】



【ヤ　行】



【ラ　行】



【ワ　行】



【欧　文】



【数　字】

## 著者略歴

つるみ　けいいち
鶴見　恵一

1948年　群馬県に生まれる

1970年　アマチュア無線機の開発に従事

現　在　自動検査装置の設計・開発に従事

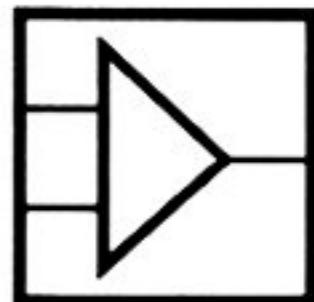
著　書　6809オールマイティ（共著　CQ出版社）

　　　　トランジスタ技術SPECIAL No.2（CQ出版社）

# 6809マイコン・システム設計作法

昭和62年7月20日　初版発行



著　者　鶴　見　恵　一
発行人　飛　坐　博
発行所　CQ出版株式会社
東京都豊島区巣鴨1－14－2（〒170）
電話 03（947）6311（代）振替 東京0-10665

定価1,500円

乱丁，落丁本はお取り替えします
ISBN4-7898-3206-6 C3055 ¥1500E

印刷・製本　図印刷（株）